VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Detekce zranitelných a zastaralých závislostí programů

Detection of Vulnerable and Outdated Dependency Programs

2017 Bc. Lukáš Raška

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Lukáš Raška**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Detekce zranitelných a zastaralých závislostí programů
Detection of Vulnerable and Outdated Dependency Programs

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je vytvořit systém pro detekci závislostí programů v rozličných programovacích jazycích. Detekci lze rozdělit do dvou kategorií:
1. Detekce zastaralých závislostí - kontrola oproti repozitářům - Maven, RubyGems, NuGet, PyPI, SBT,npm, aj..
2. Detekce zranitelných závislostí:
- bytecode analysis,
- binary analysis,
- kontrola oproti zvěřejněným CVE (nvd.nist.gov).

Základním požadavkem je možnost snadné a rychlé detekce zranitelností, modularita, škálovatelnost, snadná rozšiřitelnost a napojení na externí systémy.

Seznam doporučené odborné literatury:

[1] MCDONALD, John, Mark DOWN a Justin SCHUH, 2006. The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities. 1 edition. B.m.: Addison-Wesley Professional.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Kožusznik, Ph.D.**

Datum zadání: 01.09.2016

Datum odevzdání: 28.04.2017

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 28. května 2017 .....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 28. května 2017 .....

Rád bych poděkoval Ing. Janu Kožusznikovi, Ph.D. za cenné rady při návrhu a tvorbě této práce.

Abstrakt

Obsahem této diplomové práce je návrh a implementace platformy pro detekci závislostí aplikací a jejich následnou klasifikaci z hlediska provozního rizika. Teoretická část práce popisuje motivaci a techniky detekce závislostí, společně s problémy implementace moderních enterprise aplikací. Zároveň jsou zde popsány i metody vývoje škálovatelných platforem s ohledem na rozsáhlost dat a nasazení. Praktická část popisuje implementační detaily platformy, problémy spojené s vývojem, jakožto i obecné pokyny a návody k nasazení daného typu aplikací, obzvláště s ohledem na konfigurační management.

Klíčová slova: diplomová práce, závislosti, škálovatelnost, Java, Spring, Maven, CVE, npm, zranitelnosti, microservices, messaging, IAM, OpenID Connect, OAuth 2

Abstract

The aim of this master's thesis is to design and implement platform for application dependency analysis and classification by operational risk. First part of thesis describes the motivation and possible techniques for dependency detection, along with trends in modern enterprise application development and techniques usable for modularity and scalability, especially with concern to big data and continuous deployment. Second part of thesis describes implementation details, difficulties during implementation and general advices for developing and deploying similar technology stack.

Key Words: master thesis, dependency, scalability, Java, Spring, Maven, CVE, npm, vulnerability, microservices, messaging, IAM, OpenID Connect, OAuth 2

Obsah

Seznam použitých zkratek a symbolů

AD	–	Active Directory
API	–	Application Programming Interface
CI/CD	–	Continuous Integration, Continuous Delivery / Deployment
CVE	–	Common Vulnerabilities and Exposures
DOM	–	Document Object Model
DoS	–	Denial of Service
DSL	–	Domain Specific Language
IaaS	–	Infrastructure as a Service
IAM	–	Identity and Access Management
IdP	–	Identity Provider
IoT	–	Internet of Things
JSON	–	JavaScript Object Notation
JVM	–	Java Virtual Machine
JWT	–	JSON Web Tokens
LDAP	–	Lightweight Directory Access Protocol
MQ	–	Message Queue
PaaS	–	Platform as a Service
REST	–	Representational State Transfer
SaaS	–	Software as a Service
SAML	–	Security Assertion Markup Language
SPA	–	Single Page Application
SSO	–	Single-Sign-On
TLS	–	Transport Layer Security
UX	–	User Experience
XML	–	Extensible Markup Language

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Cílem této práce je vytvoření funkční platformy pro detekci závislostí a jejich následnou klasifikaci, obzvláště s ohledem na možné provozní riziko. Technologie postoupila do doby, kdy je velmi snadné vyvíjet aplikace za použití rozličných frameworků a obzvláště v dnešním kontextu *Internetu věcí* by měl být dbán velký důraz na bezpečnost využívaných aplikací. S růstem cloudových služeb[1] existuje spousta služeb přímo dostupných z internetu, ať již cíleně, či pouze z důvodu nedostatečného pochopení problematiky, a trend útoků na dostupné aplikace každým rokem roste. Tato práce se nevěnuje metodám zabezpečení aplikací (pouze do jisté urovně v oblasti autentizace), taktéž zde nejsou do hloubky probírané metody zabezpečení podpůrných aplikací a operačního systému, oproti tomu se věnuje problematice méně známé. S existencí podpůrných knihoven má vývojář efektivnější metody vývoje, nemusí se starat o implementaci elementárních algoritmů či známých problematik, což s sebou nese jistou iluzi bezpečnosti. V roce 2016 bylo v existujících aplikacích a knihovnách nalezeno 7470 zranitelností[2], v prvních čtyřech měsících roku 2017 to bylo již více než 1850 zranitelností [3]. Přestože jsou do statistik zahrnuty i koncové a neveřejné aplikace, existují ale i závažné zranitelnosti v používaných podpůrných aplikacích, například v kryptografických knihovnách[2], problémy týkající se serializace[3], a další. Tyto zranitelnosti mohou v lepších případech skončit nedostupnosti aplikace (tzv. DoS), v horším s vyzrazením interních dat aplikace a přidružených dat na cílových serverech. Detekci daných zranitelností lze moderně řešit statickou analýzou kódu, fuzzingem, nebo například manuální revizí kódu. Jak lze ale pohodlně zjistit, zda má aplikace využívá některou ze zasažených knihoven? Zde je tedy cílem vytvořit platformu, která umožní tuto potřebnou analýzu a to v uživatelsky přívětivé formě.

S rostoucí popularitou frameworků jsou nezbytné knihovny skryty, tranzitivní závislosti nejsou viditelné a enormní množství aplikací stále využívá staré a zranitelné verze knihoven. Například čistá aplikace v aktuální verzi populárního frameworku *Spring*, pouze s podporou REST služeb a přístupu k databázi, obsahuje 67 závislostí, z toho 21 označených jako zastaralých[4]. Pro podnikání založené na internetovém produktu nesmí existovat situace, kdy by byla v produkčním prostředí nasazena jakkoliv zranitelná verze daného produktu. Pokud je produkt open-source, existují platformy provádějící zdarma analýzu závislostí[5] a komplexní analýzu celé aplikace včetně kalkulace rizika[6], nicméně cena analýz je poměrně vysoká, nehledě na problémy při tzv. *on-premise* nasazení[7]. Vzhledem k nedostupnosti takovéto open-source platformy byla vytvořena aplikace *OpenDVS*[8] a to v jazyce Java, postavena na frameworku Spring Boot.

[1] IaaS, PaaS, SaaS
[2] CVE-2014-0160, OpenSSL, zranitelnost Heartbleed
[3] Apache Commons Collections, COLLECTIONS-580, CVE-2015-4852, CVE-2014-0160
[4] Spring Boot 1.5.2, analýza provedena přes vyvinutou platformu *OpenDVS 0.1.0*
[5] Např. *Gemnasium*
[6] Placené *JFrog Xray* či *Black Duck Hub*
[7] Nasazení aplikace na vlastních serverech, včetně uchovávaní dat
[8] Open Dependency Vulnerability Scanner

2 Požadavky na platformu

Cílem platformy není pouze poskytnout čistou analýzu závislostí aplikace, ale v kontextu *DevOps* a *CI/CD* také nabídnout kontinuální analýzu aplikací a to plně automatizovaně. Pro zaměření na co nejširší spektrum uživatelů nelze opomenout korporátní sféru, je tedy nutností umožnit nasazení celé platformy v rámci infrastruktury služby (z důvodu průmyslových standardů[9] či platných zákonů). Je také nutné vydat dílo pod nerestriktivní OSS licencí, která toto použití nijak neomezuje[10]. Pokud je v plánu využití platformy pro analýzu většího množství aplikací, je třeba brát v úvahu možnosti škálovatelnosti, s tím spojené metody nasazení aplikace a konfigurační management.

Základní požadavky na platformu jsou tedy následující:

- Snadná detekce rizikových závislostí *REQ_1*
- Detekce závislostí balíčkovacích systémů *REQ_2*
- Detekce přibalených závislostí *REQ_3*
- Modularita *REQ_4*
- Škálovatelnost *REQ_5*
- Snadná rozšířitelnost *REQ_6*
- Napojení na externí systémy *REQ_7*

Po rozdrobení základních požadavků vznikly tyto dodatečné požadavky[11]:

- Detekce balíčkovacího systému Maven *REQ_2 -> REQ_8*
- Detekce balíčkovacího systému Npm *REQ_2 -> REQ_9*
- Detekce balíčkovacího systému PyPi *REQ_2 -> REQ_10*
- Detekce balíčkovacího systému RubyGems *REQ_2 -> REQ_11*
- Detekce Java knihoven (*JAR, WAR, EAR*) *REQ_2 -> REQ_12*
- Detekce staticky linkovaných knihoven *REQ_2 -> REQ_13*
- Nahrávání aplikace přes webový prohlížeč *REQ_1 -> REQ_14*
- Analýza zdrojového kódu z Git repozitáře *REQ_1, REQ_7 -> REQ_15*

[9] Například *PCI-DSS* pro finanční sektor

[10] Je vhodnější zvolit méně restriktivní *Apache Licence 2.0* či *MIT licenci* před restriktivní *všeobecnou veřejnou licencí GNU*

[11] *X -> Y* označuje vznik požadavku *Y* ze základního požadavku *X*

- Detekce zastaralosti závislosti *REQ_1 -> REQ_16*
- Detekce zranitelnosti závislosti *REQ_1 -> REQ_17*
- SSO autentizace *REQ_7 -> REQ_18*

2.1 Detekce rizikových závislostí

Prvotním problémem při návrhu architektury a implementaci je detekce a následná klasifikace závislostí. Jelikož závislosti rozličných balíčkovacích systémů mohou mít rozličné signatury, je třeba jednoznačně definovat unikátnost napříč platformou. Například Maven závislosti jsou identifikovány pomocí identifikátoru skupiny (*groupId*) a identifikátoru artefaktu (*artifactId*), zatímco Npm závislosti jsou identifikovány pouze pomocí identifikátoru artefaktu. Mimo vlastní název závislosti zde hraje roli také verze (opakovaný výskyt stejné verze lze sloučit) a rozsah využití (*scope*)[12]. Při prvotní detekci je velmi důležité zvolit dostatečně unikátní identifikátor, ze kterého je snadné získat potřebné informace pro následnou analýzu. Taktéž je nutné, aby všechny kroky analýzy o definovaném formátu věděly a bylo možné zpětně rekonstruovat a analyzovat celý proces detekce. Pro jednoznačný identifikátor byla zvolena konvence *<skupina>:<identifikátor artefaktu dle verze>*, zatímco verze či rozsah využití budou uloženy v oddělených polích a následné shlukování bude provedeno za běhu v případě potřeby.

Pro získání relevantních informací o analyzované aplikaci je taktéž vhodné získat kompletní informace o všech závislostech. Vzhledem k povaze dat poskytují balíčkovací systémy export grafového znázornění závilostí (včetně tranzitivních závislostí), tyto data je možné následně vizuálně zobrazit. Problémem zde může být nezjištění kompletního grafu závislostí, například v případě neznámého artefaktu, obzvláště v případě nevhodného přístupu balíčkovacího systému[13].

V případě detekce nekompletních závislostí či jednotlivých částech (např. část knihovny operačního systému) není možné jednoznačně identifikovat zdroj pouze z informací poskytnutých aplikací (není v této době k dispozici kompletní běhové prostředí, je prováděna pouze statická analýza) a je tedy nutné tyto detekované závislosti ověřit oproti známé databázi. Pro tento účel se nejlépe volí vytvoření signatury souboru (hash), pro jednoduchost a kompatibilitu s externími systémy[14] byla zvolena kryptografická hašovací funkce *SHA-1*, nicméně vzhledem k nedávnému prolomení této šifry[4] by bylo vhodnější generovat i hash dle šifry *SHA-256*, která je odolnější vůči vzniku kolize.

V průběhu detekce je velmi důležité myslet na dynamické generování analyzovaných dat. Například Java archiv[15] je interně reprezentován jako ZIP archiv, lze jej tedy rozbalit a interní obsah následně analyzovat. Java archiv pro webovou aplikaci následně obsahuje adresář *WEB-INF/lib*, který obsahuje další Java archivy. Při návrhu architektury detekčních sond je třeba

[12] Rozsahem využití je myšlen konkrétní stav aplikace - kompilace, testování, běhové prostředí, aj.

[13] Například *Tree* Mojo v *Maven Dependency* pluginu zhavaruje v případě nezjištění informací o závislosti

[14] Balíčkovací systémy velmi závisí na jedné šifře, bylo by tedy nutné všechny pro ověření všechny externí závislosti stahovat a hashovat

[15] JAR, WAR, aj.

myslet na problémy zacyklení či opakované detekce stejné závislosti. Pro vyřešení možných problémů je možné například u každé sondy uchovávat informaci o analyzovaných souborech a vyvarovat se tak opakovanému zpracování či zacyklení. Tento přístup nicméně znamená uložení velkého množství dat v paměti procesu, zejména v případě rozsáhlé adresářové struktury, je tedy efektivnější zvolit extrakci nově analyzovaných dat do oddělených adresářů a uchovávat pouze odkazy na tyto adresáře.

Jakmile detekce závislostí skončí, je možné postoupit do fáze detekce metadat. Pro každou nalezenou závislost je důležité zjistit informace, které mohou být relevantní pro zjištění reálného stavu aplikace a provozního rizika. Při definování rizika dané aplikace je důležité porovnávat rozličné atributy závislosti, jmenovitě:

- Detekovaná vs. aktuální verze
- Datum vydání
- Rozsah využití
- Známé zranitelnosti

Při porovnávání detekované verze je nutné brát v úvahu vývoj v několika etapách. Aplikace může udržovat stabilní verzi, která se zcela jistě bude lišit od nejnovější verze. Zde vzniká problém v případě balíčkovacích systémů, které neumožňují označit více verzí jako aktuální. V případě, že vývoj aplikace dodržuje sémantické verzování[16], je možné implementovat jistá pravidla, například za pomoci data vydání, nicméně vzhledem k ohromnému množství aplikací není možné manuálně vkládat informace o aktuálních verzích a tedy zajistit bezchybnou analýzu. Také využití závislosti v běhovém prostředí aplikace bude mít vyšší vliv na riziko než závislost použita pouze v době testování.

Velkým problémem je detekce známých zranitelností. Dle veřejné *National Vulnerability Database*[17] je možné zjistit kolik a jaké oznámené zranitelnosti existují, lze také zjistit popis či další informace, ale jednoznačný název problémové komponenty zde dostupný není. Ze zmíněné databáze lze v určitých případech jednoznačně získat název zranitelné aplikace[18], nicméně nejednoznačnost pojmenování může být velmi zavádějící. Například knihovna s paralelní podporou více programovacích jazyků může trpět zranitelností ve variantě A, nicméně detekce zjistí i verzi B. Totožně může vzniknout chybná detekce v případě špatného pojmenování, například JDBC knihovna pro MySQL zachovává naprosto odlišné verzování od MySQL databáze a zranitelnosti vzniklé v dané databázi mohou být detekovány i pro zmíněnou knihovnu. Tento problém z části řeší jednotlivé operační systémy, které vydávají bezpečnostní bulletin o objevených zra-

[16] SemVer 2.0.0

[17] *http://nvd.nist.gov/*

[18] V případě placené VulnDB je šance správné detekce vyšší

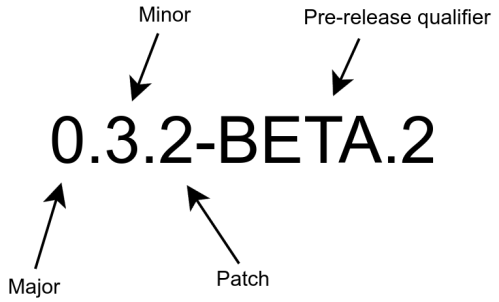
nitelnostech distribučních balíčků a zároveň poskytují přehledné API pro získání podrobnějších informací[19].

2.2 Sémantické verzování

V případě vývoje aplikace je unifikace procesů ohromnou výhodou. Pokud pro konkrétní proces existuje definovaný standard, lze předpokládat, že je standard otestován v praxi a řeší širokou škálu problémů. Pro verzování aplikací existuje sémantické verzování, de-facto standard, kdy můžeme pouhým porovnáním verzí odvodit, jak moc velké změny byly v aplikaci provedeny. Sémantické verzování ve verzi 2.0.0 definuje několik pravidel, která by měla, či musí být dodržena při vývoji aplikace[5]:

1. Verze aplikace musí mít formát *<MAJOR>.<MINOR>.<PATCH>* s volitelným modifikátorem
2. *MAJOR* číslo značí vznik nekompatibilního API
3. *MINOR* číslo značí vznik zpětně nekombatibilní změny / větší změny / zastaralost části API
4. *PATCH* číslo značí zpětně kompatibilní opravu chyb
5. Jakmile je verze vydána, obsah se nesmí změnit

Kromě zmíněných pravidel existuje ještě řada pravidel a doporučení, například verze 1.0.0 značí první stabilní verzi aplikace. Při dodržení dané specifikace můžeme zajistit jednoznačnost verzí a dávame uživateli záruku, například stabilitu vystaveného API.



Obrázek 1: Struktura verze při sémantickém verzování

[19]Například https://security-tracker.debian.org

2.3 Zpracování balíčkovacích systémů

Aplikační balíčkovací systémy umožňují při vývoji aplikací definovat všechny známé závislosti a úkolem daného systému je dodání balíčku ve formě přijatelné pro danou platformu. Zatímco v případě komplexnějších aplikací a platforem plní úlohu balíčkovacího systému nástroj určený pro sestavování aplikace (například Maven), ne vždy je toto podmínkou. V případě jazyků, které mají jasně definovaný distribuční a spouštěcí mechanismus lze předpokládat, že má daná platforma zajištěnu distribuci závislostí společně s aplikací (například statické linkování binárních knihoven, Java archivy). U zbylých jazyků a platforem máme povětšinou zajištěnou uniformitu závislostí v rámci operačního systému či běhového prostředí uživatele[20] a existuje zde jasně definovaný formát distribuce informací o závislostech. Z výše zmíněných pravidel lze vydedukovat podmínky pro implementaci libovolného balíčkovacího systému:

1. Jasná struktura definice závislostí
2. Možnost získat tranzitivní závislosti[21]
3. Definované API pro získání metadat artefaktu

2.4 Detekce Java knihoven

Jak bylo již výše zmíněno, Java knihovny jsou interně reprezentovány jako ZIP archivy. Pro základní zpracování lze tyto knihovny pouze rozbalit, vykalkulovat hash souboru a pokračovat v detekci ostatních závislostí[22]. V případě nutnosti detailnější analýzy lze detekovat signaturu balíčku dle definované struktury:

- */META-INF/MANIFEST.MF*
- */WEB-INF/*
- */WEB-INF/lib/*

Výše zmíněné složky a soubory mohou obsahovat užitečné informace vedoucí k jednoznačné identifikaci balíčku, nicméně to není vyžadováno[6]. Soubor *MANIFEST.MF*, dostupný ve většině správně vytvořených balíčků, může obsahovat metadata vzniklá při sestavení aplikace, lze tedy v některých případech odvodit Maven identifikátor. Ostatní zmíněné složky slouží k uskladnění potřebných zdrojů pro webové aplikace, tedy nutné závislosti, konfigurační soubory, šablony, aj. Tyto soubory již nejsou z hlediska identifikace velmi zajímavé, nicméně pro detekci dodatečných závislostí je dobré je analyzovat.

[20] Např. Python, RubyGems

[21] Není nutnou podmínkou, uživatel nicméně nedostane úplná data

[22] Tento krok je nutným základem, knihovny mohou obsahovat další knihovny v rozličných tvarech - další Java knihovny, systémové knihovny, aj.

2.5 Detekce staticky linkovaných knihoven

Při sestavování aplikací se staticky linkovanými knihovnami dochází k zahrnutí všech závislostí do výsledného artefaktu. V případě, že nejsou do artefaktu zahrnuty potřebné značky, je téměř nemožné jednoduše detekovat všechny závislosti. Pokud výsledná aplikace obsahuje vývojářské značky, lze detekovat symboly, které mohou pocházet z jiných knihoven, například pomocí aplikace *nm*, dostupnou v sadě aplikací *GNU Binutils*. Pro detekci přidružených knihoven je možné následně tyto symboly porovnat s databází a tím zjistit danou knihovnu. Problémem je také detekce verzí, jelikož signatura funkcí nemusí být mezi verzemi rozdílná.

```
                 U accept@@GLIBC_2.2.5
                 U bind@@GLIBC_2.2.5
0000000000607320 B __bss_start
                 U calloc@@GLIBC_2.2.5
                 U chdir@@GLIBC_2.2.5
                 U close@@GLIBC_2.2.5
                 U SSL_accept@@libssl.so.10
                 U SSL_CTX_new@@libssl.so.10
                 U SSL_CTX_use_certificate_chain_file@@libssl.so.10
                 U SSL_CTX_use_PrivateKey_file@@libssl.so.10
                 U SSL_library_init@@libssl.so.10
                 U SSL_load_error_strings@@libssl.so.10
                 U SSL_new@@libssl.so.10
                 U SSL_read@@libssl.so.10
                 U SSL_set_fd@@libssl.so.10
                 U SSL_shutdown@@libssl.so.10
                 U SSLv23_server_method@@libssl.so.10
                 U SSL_write@@libssl.so.10
0000000000402a6b T _start
0000000000607328 B stderr@@GLIBC_2.2.5
                 U strcmp@@GLIBC_2.2.5
                 U strdup@@GLIBC_2.2.5
                 U strerror@@GLIBC_2.2.5
                 U strtol@@GLIBC_2.2.5
                 U syslog@@GLIBC_2.2.5
                 U __sysv_signal@@GLIBC_2.2.5
0000000000607320 D __TMC_END__
                 U vsnprintf@@GLIBC_2.2.5
                 U waitpid@@GLIBC_2.2.5
                 U write@@GLIBC_2.2.5
```

Výpis 1: Část symbolů open-source aplikace

Pokud chceme zohlednit placené nástroje pro detekci závislostí, existuje široká škála dekompilátorů a analyzátorů, které dokážou mimojiné i detekovat jisté závislosti[23]. Bohužel v případě vývoje platformy jako OSS není vhodné nabízet funkcionalitu, která je vázána na specifickou proprietární aplikaci. Z tohoto důvodu nebyla detekce staticky linkovaných knihoven zahrnuta do plánu diplomové práce.

2.6 Modularita a microservices

Při vývoji platformy je velmi důležitý návrh celkové architektury systémů, včetně definování nezbytných integračních závislostí. Pokud je vyvíjena aplikace s rozsáhlou business logikou, je vhodné jednotlivé části separovat do oddělených komponent, mezi kterými bude komunikace probíhat přes jasně definované API. Aktuálním trendem v tomto ohledu jsou microservices, jednoduše řečeno se jedná o logickou separaci jednotlivých komponent do oddělených aplikací. Tyto aplikace mezi sebou komunikují převážně přes webové API, či alternativně přes *service bus*. Z pohledu enterprise architektury se tedy jedná o rozdělení komponent do jednotlivých *Tiers*. Komplikací při volbě libovolného architektonického vzoru je důležité daný vzor správně pochopit. Vzhledem k velkému trendu microservice v aktuální době existuje velká spousta návodů či implementačních ukázek, které zásadně porušují pravidla tohoto vzoru. Mezi problémy, které musí programátoři a architekti řešit, patří převážně jasná specifikace komunikačního kanálu a intuitivní návrh architektury. Velmi často se lze v praxi setkat s aplikacemi, které jako monolitická aplikace vyžadují malé množství zdrojů, zatímco v kontextu microservices je aplikace rozdrobená do desítek jednotlivých komponent a zdrojů vyžadují mnohem více.

Microservices mají nicméně ohromné výhody v nasazování a udržování prostředí, bezpečnosti či škálování. V případě vzniku chyby v jednotlivé komponentě lze velmi snadno vyměnit verzi komponenty za jinou a ostatní části aplikace nebudou výpadkem zasaženy. Taktéž v případě nalezení úzkého hrdla je možné škálovat pouze danou komponentu, pokud to implementace dané architektury podporuje. Při dekompozici aplikace do komponent je vhodné brát v potaz business doménu a velikost business logiky[7], obzvláště s ohledem na možnou velkou duplikaci logiky.

Pro zajištění modularity je velmi vhodné co nejvíce cílit na microservices architekturu, je ale velmi důležité zvážit skutečnou potřebu vývijené aplikace. Jelikož žádná architektura či návrhový vzor není univerzální, není možné zajistit jednolitost aplikace a vhodněji se zde nabízí hybridní architektura, použita i tomto projektu.

[23] např. IDA F.L.I.R.T.

2.7 Škálovatelnost

Při návrhu architektury systému je třeba brát v potaz předpokládanou zátěž, obzvláště v dnešním kontextu cloudových služeb. Jestliže jedním z cílů je nabízet produkt koncovým zákazníkům, nevyplatí se vytvářet oddělené instance pro potencionálního zákazníka, ale využití jedné, sdílené instance, je mnohem efektivnější. Jestliže aplikaci stavíme na microservices architektuře, je nalezení a naškálování úzkého hrdla aplikace mnohem jednodušší. Při návrhu škálovatelného systému je třeba brát v potaz několik kritických okruhů:

- Komunikační protokol
- Souběh
- Rozkládání zátěže (load-balancing)
- Integritu dat
- Blokující vs neblokující operace

Pro komunikaci mezi jednotlivými částmi aplikace lze obecně využít přímé spojení na definované API. Pokud pouze přeposíláme požadavky, či vyžadujeme okamžitou odpověď, je tento princip bezproblémový, obzvláště při vhodně nakonfigurovaném load-balancingu. V případě, kdy chceme akci asynchronně vykonat, tento princip není velmi vhodný a vzniká s ním spousta otázek:

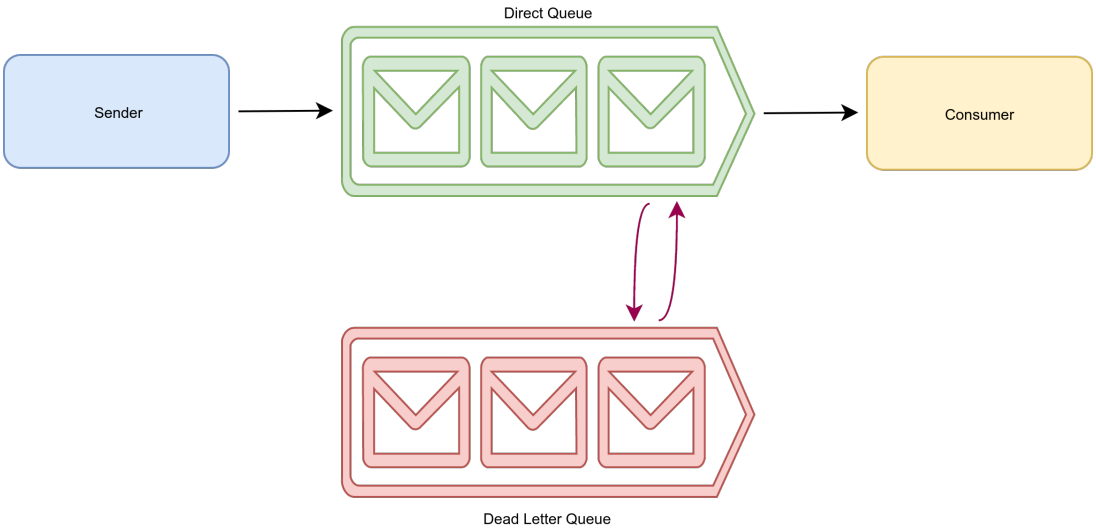
- Jak víme, která komponenta je nejméně zatížena?
- Jak opakovaně zpracovat nedokončenou akci?
- Jak zjistit, zda prostředí není poddimenzované?
- Jaké bude chování při nárazové zátěži?

Pro první otázku je možné nadefinovat speciální API volání. U druhé otázky můžeme doufat, že komponenta má ošetřeny všechny nežádoucí stavy a nedojde k výpadku integrovaných systémů. U třetí otázky můžeme využít monitoring a pro otázku čtvrtou můžeme implementovat frontu zpracovávaných akcí. Jestliže se bavíme o architektonických vzorech a komponentovém rozdělení, neměli bychom zapomínat na využití již existujících implementací, knihoven či aplikací. Pro tento typ problému existují *fronty zpráv (MQ)*[24], které dávají odpověď na uvedené otázky. Při použití externí aplikace pro MQ máme zajištenu integritu zpráv, doručenost a zároveň i jedno definované API pro zjištění aktuálního stavu zpracovávání / provádění akcí.

[24]Například RabbitMQ (AMQP), ActiveMQ (JMS), ZeroMQ, aj.

2.7.1 Message queue

Použitím MQ se vyvarujeme řadě problémů, nicméně je zde třeba myslet hlavně na problém souběhu. Nesmíme zapomínat, že vykonávané akce jsou prováděny asychronně, nemáme zajišteno zpracování zpráv stejnými uzly, také je nutné pečlivě myslet na vysokou paralelizaci řešení. Odesílání zpráv, pro které je vyžadováno sériové zpracování, není pro MQ vhodné a existují lepší architektonické vzory. Při generování zpráv systému pracujícího s jednou databázovou instancí je také důležité správně zpracovávat transakce. Jestliže odešleme zprávu, která pracuje s entitou ukládanou do databáze, je nutné se ujistit, že je entita v databázi uložena a nenacházíme se v transakčním kontextu. V opačném případě může dojít ke zpracovávání entity na komponentě, která do transakce jiné komponenty přístup nemá a entita nemusí být ve správném stavu. Také je vhodné při zpracovávání zprávy využít transakcí na úrovni MQ systému a v případě neplatné operace či chování vyhodit vyjímku. Při správně nastavené *Dead Letter Queue* bude zpráva přesunuta do fronty označené jako chybové a zpráva nebude ztracena. Tuto zprávu lze po vyřešení vzniklého problému (např. implementační chyba komponenty) přesunout do fronty původní a následně bude znovu zpracována.

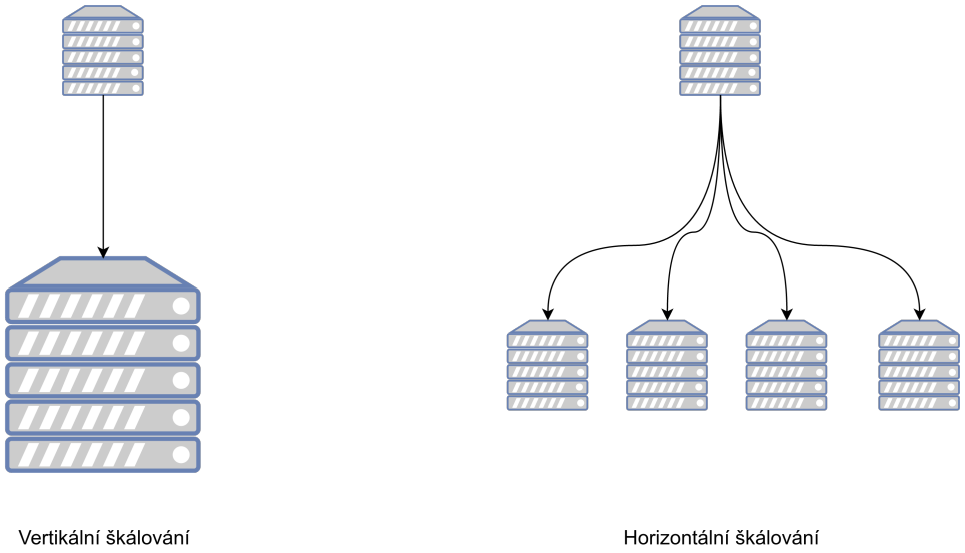


Obrázek 2: Princip Dead Letter Queue

2.7.2 Load balancing

Pro nastavení load balacingu existuje několik metod. Nejčastěji používanou metodou je tzv. *Round-robin*, tedy férové přerozdělování operací. Ve dvouuzlové konfiguraci bude první operace předána uzlu 1, druhá operace uzlu 2, třetí operace opět uzlu 1 a takto pořád dokola. Variantou k Round-robin load-balancingu je *least connected* metoda, která bude vždy předávat operace uzlu s nejméně probíhajícími operacemi. Obě zmíněné metody (a jejich variace) nicméně vyžadují bezestavovost koncového systému. V případě, že je třeba dodržet kontext, je nutné operaci označit, aby load-balancing byl schopen na základě této značky správně směrovat požadavek.

V kontextu síťových a webových služeb je například možné směrovat TCP požadavky pomocí zdrojové IP adresy, HTTP požadavky pomocí hlaviček či cookies. S ohledem na škálovatelnost je nutné se stavovosti zbavit, v opačném případě nelze koncový systém efektivně horizontálně škálovat. Další možností značení požadavků je uchovávat mapování přímo na straně load-balanceru, nicméně tato metoda vyžaduje efektivní synchronizaci celého LB systému (informace je nutné vyměnit synchronně, aby bylo zajištěno uniformní směrování klienta), což může mít za následek snížení propustnosti celého řešení.



Obrázek 3: Horizontální a vertikální škálování

2.8 Externí systémy

Důležitou podmínkou platformy je snadná oboustranná integrace s různými systémy použitými při vývoji software. V případě kontinuální detekce produkční verze aplikace je vhodné například notifikovat administrátory (email, Slack, aj.), či rovnou vytvořit záznam v *Issue tracking systému*[25]. Z opačné strany je efektivnější provádět analýzu na základě definovaného vstupu či notifikace (*push* přístup), nežli periodicky kontrolovat, zda máme akci provést (*polling* přístup). Je tedy nutné vydefinovat dostupná API pro oboustrannou komunikaci.

Pro příchozí notifikaci z aplikace třetí strany zde již máme definován jeden API endpoint - nahrání analyzované aplikace, stačí tedy tento přístup rozšířit o explicitní spuštění analýzy v případě jiného zdroje dat. Je také důležité myslet na bezpečnost, analýza může být spuštěna automatizovaně, například z CI systému, nelze tedy provádět autentizaci přímo pomocí uživatelských údajů[26] a je důležité definovat alternativní identifikaci uživatele. Zde je možné uživateli nabídnout generování API tokenů, například s jistým omezením (doba platnosti, rozsah použití). Druhotným implementačním detailem by zde mohla být modularita notifikačních API endpointů

[25] Jira, Bugzilla, či ITIL-compliant systémy
[26] Nehledě na razantní konflikt při použití IdP s confidential klientem

pro zpracování informací z obsahu požadavku, ve kterých se v jistých případech může vyskytovat odkaz na stáhnutí artefaktu. Vhodnou volbou by tedy bylo nechat implementaci zmíněných endpointů v režii modulu pro konkrétní typ zdroje dat.

Pro odchozí notifikaci je opět vhodné definovat jasně specifikované rozhraní, aby bylo možné dynamicky přidávat další typy notifikací. Jelikož v tomto případě nedochází k poskytnutí tohoto rozhraní třetím stranám, je zde implementace jednodušší a je v tomto ohledu nutné pouze myslet na možnou změnu vzdáleného API a na asynchronní provádění daných notifikací. Prvotní notifikací, a často nezbytnou v případě SPA, zde bude notifikace uživatele pomocí technologie *WebSocket*, která umožní notifikovat uživatele v prohlížeči téměř v reálném čase s minimálními požadavky na zdroje[27]

2.9 User experience

Důležitým aspektem při vývoji klientských aplikací je uživatelská přívětivost neboli *user experience*. Narozdíl od rozšířeného omylu, UX není pouze vzhled, ale zahrnuje veškeré kroky interakce uživatele. Rychlost odezvy, rozložení prvků, vazby mezi entitami i klávesové zkratky přispívají k zlepšení uživatelské přivětivosti celého systému. Při návrhu UX je vhodné vyřešit několik okruhů:

- Návrh entit
- Návrh rozhraní
- Reaktivnost a zpětná vazba
- Odezva

Pro implementovanou platformu byly definovány následující modely: *projekt*, *artefakt* a *komponenta*. Projekt se skládá z artefaktů, které jsou definovány sadou komponent. Každý projekt má taktéž definován vlastní typ (např. Git), který určuje zdroj dat.

Díky koncipování platformy jako webovou aplikaci lze využít z několika existujících frameworků pro návrh rozhraní. Zde je pravděpodobně nejpoužívanějším frameworkem projekt *Bootstrap*[28], mezi alternativy patří například framework *Foundation*, který sází primárně na responzivnost obsahu. Tyto frameworky kladou důraz na definici jednotlivých prvků používaných při návrhu webových stránek, což způsobuje i jistý problém. Přestože při implementaci frameworků je brán důraz na responzivní design, tedy přehledný a intuitivní vzhled stránky na širokém spektru zařízení, problémem zde jsou mobilní aplikace. Nativní mobilní aplikace umožňuje využívat všech výhod daného zařízení, prohlížeč oproti tomu obsahuje velkou řadu omezení, obzvláště z důvodu bezpečnosti. Unifikovanost ovládání a vzhledu velkého množství

[27] WebSocket vytváří pouze jedno TCP spojení, které je udržováno pomocí keepalive zpráv. Princip je tedy velmi podobný *publish-subscribe* přístupu.

[28] V době psaní této práce je Bootstrap druhým nejoblíbenějším projektem hostovaným na serveru GitHub

aplikací taktéž vytváří jistý standard, nové aplikace postavené na tomto standardu jsou pro uživatele přístupnější, jednodušší na používání a jejich chování je předvídatelnější. Z tohoto důvodu v roce 2014 vydala společnost *Google* studii, kde představili nový grafický jazyk nazvaný *Material Design*, který detailně popisuje jednotlivé typy elementů používaných v rozličných aplikacích a taktéž jejich rozsah použití a chování. Na základě Material Designu vznikla řada knihoven, které poskytují vývojářům snadnější adaptaci daných princpů.

I přes definování elementů a *wireframing* je zde celá řada faktorů, která ovlivňuje uživatelskou přívětivost. Velmi důležitým faktorem je obzvláště odezva rozhraní, respektive reaktivnost (schopnost a efektivnost odezvy na akci). Při standartním vývoji webových aplikací je stránka složena z uspořádání elementů (HTML), které jsou zvýrazněny pomocí kaskádových stylů (CSS) a pro dynamičnost je zde používán JavaScript. Tento princip je poměrně neefektivní i při vývoji jednodušších aplikací, příkladem může být posloupnost akcí při uživatelském vstupu či využití stejné hodnoty v rozdílných částech aplikace. Dalším, ne již tak razantním problémem, je ztráta aplikačního kontextu v případě změny stránky[29]. Tento problém řeší *SPA*, tedy *Single Page Application*, kde je aplikace složena ze sady šablon a jakékoliv akce, včetně zobrazení jiné sekce (*routing* princip), jsou obstarány pomocí JavaScriptu. První, důležitější problém, je zde řešen pomocí obrácení logiky, tedy jednotlivé elementy stránky nejsou obohaceny o JavaScript, ale JavaScript je obohacen o sadu elementů, které dynamicky vykresluje. Přestože je princip *Single Page Application* poměrně starým, pochází již z roku 2003[8], popularitu získal hlavně v po roce 2010 s rozmachem JavaScriptového frameworku *AngularJS*, jež sází na princip deklarativního programování. Původní princip frameworku je již překonán a v praxi se nyní používá převážně knihovna *React* či framework *Angular* (nástupce původního AngularJS), které velmi usnadňují vývoj a testování logiky webových aplikací[30]. Již zmíněný routing, tedy směrování uživatele na správný obsah (stránku), je v SPA řešen pomocí HTML kotev, tedy principu původně myšleným pro směrování na pozici konkrétního HTML elementu.

2.9.1 Angular

Základním stavebním pilířem starší verze frameworku Angular byl tzv. *two-way binding*, tedy oboustranné mapování událostí proměnných. Jestliže je proměnná upravena (resp. je do ni přiřazena hodnota), událost je zaznamenána a daná stránka je překreslena. Přestože tento princip umožňuje velmi efektivní integritu a sdílení proměnných, nese s sebou i jistá rizika. U knihoven, které nejsou ta tento způsob připraveny (převážně obecné knihovny, např. vykreslování grafů), vzniká nekonzistence, špatné chování vzhledem k uživateli (UX) a při komplexnějších aplikacích jsou větší nároky na zdroje (CPU). Z tohoto důvodu v novějších verzích frameworku není two-way binding podporován, rozdíl je pouze při použití *ngModel* direktivy. I přesto je možné

[29] Dochází k načtení nové stránky, provedené akce na předchozí stránce jsou zahozeny, pokud nejsou uloženy na serveru

[30] Serverová část v tomto případě slouží spíše jako návrhový vzor *Repository* publikovaný přes REST API

napsat vlastní komponenty podporující tento princip, je-li to vývojářem vyžadováno. Angular 2+ taktéž doporučuje *TypeScript*[31] pro snadný vývoj.

```
<h1>{{title}}</h1>
<h2>My Heroes</h2>
<ul class="heroes">
  <li *ngFor="let hero of heroes"
    [class.selected]="hero === selectedHero"
    (click)="onSelect(hero)">
    <span class="badge">{{hero.id}}</span> {{hero.name}}
  </li>
</ul>
<hero-detail [hero]="selectedHero"></hero-detail>
```

Výpis 2: Šablona komponenty v Angular 2

2.9.2 React

Zatímco Angular je plnohodnotý framework, obsahuje tedy velkou škálu funkcionalit potřebných pro vývoj webových aplikací, React je oproti tomu pouze knihovna. Základním principem je nabídnout vývojáři pouze nezbytně nutné metody pro základní funkcionalitu webových aplikací. React tedy nabízí pouze metody pro vykreslení a životní cyklus jednotlivých komponent. V případě, že jsou vyžadovány dodatečné funkcionality, je k dispozici velká škála rozšíření (například *react-router* pro routing).

Základním principem knihovny React je *virtuální DOM*. Každá změna, která vyžaduje překreslení webové aplikace je nejdříve vykreslena virtuálně, následně jsou porovnány změny (virtuální DOM vs skutečný) a překresleny jsou pouze změny, které vznikly. Tento přístup razantně zlepšuje výkon webové aplikace (virtuální DOM je pouze nevykreslená struktura sídlící v operační paměti) a z části řeší problémy integrace nativních JavaScript knihoven (např. při změně tabulky není překreslen graf sídlící v jiném elementu). Pro zajištění správné hodnoty stavu aplikace (*state*) je ze vynucen *one-way binding* a pro změnu stavu je nutno explicitně zavolat speciální metodu. Další velkou odlišností od frameworku Angular je přístup k vykreslování elementů. Zatímco Angular obohacuje existující HTML elementy (či definované komponenty), React využívá JSX[32], kdy jsou pomocí JavaScriptu generovány konečné HTML elementy. Tento přístup umožňuje přímo využívat JavaScript při definici rozhraní a kombinovat HTML značky společně s JavaScript přístupem.

[31] Existují rozšíření pro ES5 verzi JavaScriptu či Dart, nicméně drtivá většina vývojářů využívá TypeScript, či nádstavbu AtScript vyvinutou Angular týmem

[32] https://facebook.github.io/react/docs/jsx-in-depth.html

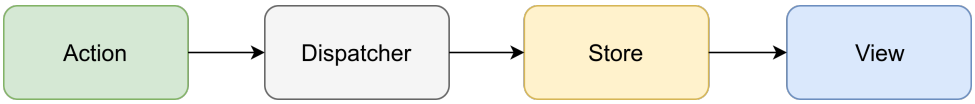
```
<Table selectable={false}>
    <TableHeader adjustForCheckbox={false} displaySelectAll={false}>
        <TableRow>
            <TableHeaderColumn>Name</TableHeaderColumn>
            <TableHeaderColumn>Type</TableHeaderColumn>
        </TableRow>
    </TableHeader>
    <TableBody>{rows}</TableBody>
    <TableFooter>
        <TableRow>
            <TableRowColumn style={{paddingTop: 20, textAlign: "center"}}>
                <IconButton disabled={page.current == 1} style={{top: 8}}
                    onClick={()=> onPageChange(page.current - 1)}>
                    <ChevronLeft />
                </IconButton>
                {pageButtons}
                <IconButton disabled={page.current == page.total} style={{top:
                    8}} onClick={()=> onPageChange(page.current + 1)} >
                    <ChevronRight />
                </IconButton>
            </TableRowColumn>
        </TableRow>
    </TableFooter>
</Table>
```

Výpis 3: Šablona komponenty v React

Z hlediska přístupu SPA aplikací není vždy snadné vytvořit dlouhotrvající stavové aplikace (například navázány na serverové sezení - session). Taktéž není vždy vhodné oddělit stavy jednotlivých komponent, například z důvodu zbytečné duplikace dat. Ze zmíněných důvodů vznikla architektura *Flux*, která jednoznačně definuje posloupnost kroků a akcí. Základním pricipem této architektury je jednosměrná modifikace dat (*unidirectional data flow* / *one-way binding*), kde definuje čtyři základní elementy:

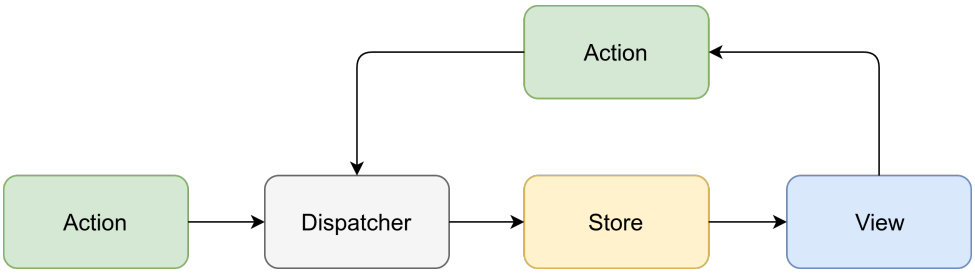
- Akce
- Dispatcher
- Store
- View

Prvotním vstupem je *Akce*, která definuje vzniklou změnu, *Dispatcher* tuto akci předá do stavových úložišť (*Store*). Zde je akce zpracována, stav aplikace je upraven a změna je zpropagována k vykreslení (prezentační vrstva - *View*).



Obrázek 4: Redux unidirectional flow

V případě, že je třeba z prezentační vrstvy změnit aplikační stav, není k úložišti přistupováno přímo, ale je vyvolána potřebná akce.



Obrázek 5: Redux unidirectional flow s klientskou akcí

Jelikož je *Flux* pouze definovaný architektonický vzor, existuje řada implementací pro rozdílné knihovny a frameworky, například *Reflux*. Pro aplikaci principu na klnihovnu *React* je nejpoužívanější knihovnou *Redux*. Redux je velmi inspirován Flux architekturou, nicméně není její plnou implementací. Zásadním rozdílem je definování jednotného aplikačního stavu (*Store*). Tato změna zásadně ovlivňuje celkovou architekturu aplikace, nicméně umožňuje mít jasně definovaný aplikační stav. Toto lze dále využít například pro serializaci stavu (uložení a obnovení stavu aplikace) či pro vývojářské potřeby[33]. Nedochází nám tedy k duplikaci dat mezi různými komponentami a máme jasně oddělenou datovou vrstvu od prezentační (princip tedy z velké části podporuje třívrstvou architekturu).

2.9.3 Nativní aplikace

V kontextu SPA aplikací není příliš složité implementovat nativní aplikaci pro rozličné zařízení. Jestliže se můžeme spolehnout na jasně definované API na backendu služby, není již velmi složité vytvořit klienta utilizujícího toto API. Jesliže nechceme zbytečně implementovat rozdílné UI (obzvláště pro zajištění jednotné UX), lze za jistých podmínek využít již implementované UI ve zmíněných technologiích. Pro React existuje sada knihoven a kompilátor *React Native*, který za pomocí stejných principů dokáže vygenerovat nativní mobilní aplikaci.

Jesliže chceme využívat webovou aplikaci i jako desktopovou, nabízí se možnost využít framework *Electron*. Princip frameworku je velmi triviální, zjednodušeně se koncová aplikace skládá

[33] Redux DevTools, které umožňují inspekci aplikačního stavu a *time travel*

z prohlížeče *Chromium* (open-source základ prohlížeče *Google Chrome*), který vykresluje námi vyvinutou webovou aplikaci a pro ucelenou integraci s koncovým systémem zde existuje řada vylepšení. Za pomoci tohoto frameworku je vytvořena velká řada známých aplikací (např. editory *Atom* či *Visual Studio Code*) i řada klientů různých služeb (*Slack*, *Mattermost*, či jiné).

3 Autentizace

Pro ověření identity uživatele v rámci korporátní sféry existuje řada rozličných systémů. Velmi rozšířeným řešením je ověření vůči AD / LDAP doméně, bylo by tedy vhodné systémy připravit na tento druh integrace. Prvotně je velmi vhodné zmínit často zaměnovaný rozdíl mezi *autentizací* a *autorizací*. Zatímco účelem autentizace je ověření uživatele, tedy že daný uživatel je skutečně danou osobou (např. ověření uživatelského jména a hesla), autorizace nám udává, zda je konkrétní uživatel oprávněn provést danou akci. Pro zjednodušení logiky je v implementované platformě prvotně implementovana autorizace v systému, pouze ve třech stavech - *žádný přístup*, *pouze pro čtení*, *plný přístup*, a to pouze pro projekty[34].

S ohledem na výše zmíněnou modulární a škálovatelnou architekturu, rozličné způsoby autentizace a webové služby, je vhodné zvolit standardizovaný protokol pro propojení s IdP. Zde se nabízí sada protokolů SAML a OAuth, obzvláště druhý zmíněný ve verzi 2.0 je využíván velkou škálou poskytovatelů identit[35] (Facebook, Google, GitHub, aj.). Pro implementovanou platformu byla zvolena integrace pomocí OpenID Connect (rozšíření protokolu OAuth2), která ma přehledněji definovaný standard a komunikace je přívětivější k SPA. Základní princip IdP je předání informací k *poskytovateli služby*[36], jako zdroj dat může sloužit například lokální databáze, či právě výše zmíněná AD / LDAP doména. Také lze často využít tzv. federaci, kdy je identita uživatele získávána z dalšího IdP. Uživatelské atributy, které lze předat poskytovatelům služeb, se označují anglickým slovem *claims*. Je důležité zmínit, že IdP může sloužit i pro autorizační potřeby, převážně se tak ale děje z obecnějšího důvodu, než by poskytovatel služby vyžadoval:

- Ověření, zda má uživatel právo na přístup do služby (např. dle uživatelských skupin)
- Ověření, zda má služba právo přistoupit k daným uživatelským atributům

3.1 SAML 2.0

Security Assertion Markup Language je standard definovaný organizací společností OASIS a jako komunikační formát využívá XML. Samotný proces identifikace uživatele má následující kroky:

1. Uživatel přistoupí na zabezpečenou stránku služby
2. Služba přesměruje uživatele k IdP s žádostí o autentizaci
3. Uživatel se přihlásí u IdP (v případě, že není přihlášen)
4. IdP vytvoří XML dokument obsahující požadovaná data, podepíše ho pomocí X.509 certifikátu a připojí k přesměrování na službu (v případě přístupu ke službě)

[34] Do budoucna je vhodné implementovat uživatelské skupiny s mapováním oproti použitému IdP
[35] S mírnými deviacemi oproti definovanému standardu
[36] Service Provider

5. Služba, znající certifikát IdP, ověří XML dokument a extrahuje potřebná data

6. Uživatel je přihlášen a může využívat aplikaci

3.2 OAuth2 a OpenID Connect

Zatímco SAML 2.0 řeší jak autorizaci, tak i autentizaci, u OAuth2 je jasně specifikován rozdíl. OAuth2 je autorizační framework, standard, určen jak pro webové aplikace, tak mimojiné i pro desktopové či mobilní aplikace. Jelikož OAuth2 neřeší žádný aspekt identifikace uživatele, pouze je zde řešena autorizace (tedy zda má uživatel přístup ke koncové službě či datům), vzniklo rozšíření OpenID Connect, které má za účel vyřešit právě tyto nejasnosti.

3.2.1 OAuth2

Jak již bylo zmíněno, OAuth2 je spíše frameworkem, než striktně specifikovaným protokolem. Definuje tedy pouze základní principy, zatímco implementační detaily se mohou lišit a z čehož plyne i možná nekompatibilita. Základním principem OAuth2 frameworku je definování čtyř základních rolí:

- Authorization server
- Klient
- Resource owner (uživatel)
- Resource server

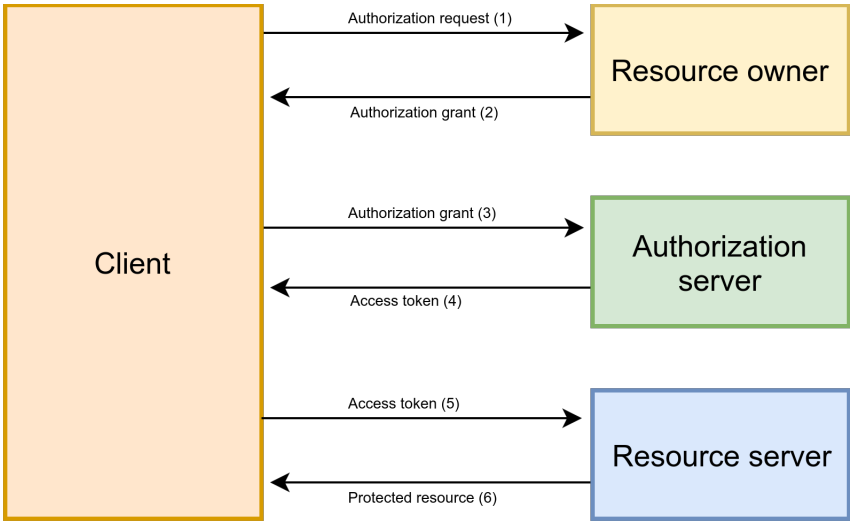
Klientem je zde myšlena aplikace, který žádá o přístup ke zdroji informací - tedy výše zmíněnému *Resource serveru*. *Resource owner* (koncový uživatel) specifikuje, či spíše poskytuje přístupová práva klientovi k danému zdroji informací. Základní ověření uživatele provádí *Authorization server* a výstupem je vydání přístupového tokenu (*access token*). Tento token lze již použít pro přístup k potřebným datům.

Jak je patrné, definice je velmi abstraktní. Také je vhodné zmínit, že rozdělení je pouze logické a *Authorization server* se velmi často vyskytuje ve stejné aplikaci, jako *Resource server*. OAuth2 také nespecifikuje požadovaný formát výměny dat, či nevyžaduje přístup k údajům uživatele[37][9]. Z ilustrace je patrné, že uživatel komunikuje s klientem, není zde nicméně nutnost předávání soukromých přístupových dat, navíc ani autorizační server tyto data nevyžaduje, je pouze nutné se domluvit na správné výměně informací. Obecně lze brát autorizační server jako aplikaci s přístupem k databázi uživatelských údajů a která provádí ověření uživatele.

Velmi častou implementací přístupových tokenů je tzv. *Bearer token*, tedy textový token [38], který je předáván v HTTP hlavičce *Authorization*. Tímto tokenem je možné se u resource serveru

[37] RFC 6749

[38] Náhodný řetězec, který by neměl být odvoditelný od žádných důležitých informací



Obrázek 6: Abstraktní OAuth2 komunikace

ověřit a získat přístup k požadovaným informacím. Je důležité upozornit, že bearer token má pouze omezenou platnost (bežně max. několik desítek sekund). Pro aplikace, které pracují se zabezpečenými zdroji v delším časovém horizontu (např. inteligentní proxy), zde existuje *Refresh token*, pomocí kterého je možné si vyžádat nový bearer token.

OAuth 2.0 také specifikuje dva typů klientů[39]:

- Confidential
- Public

Typ *confidential* vyžaduje striktní přesměrování na autorizační server, kde je uživatel ověřen, zatímco typ *public* toto nevyžaduje a uživatel může pomocí definovaného rozhraní předávat uživatelské údaje skrz klienta. Tento druhý typ je velmi často oblíbený přávě u SPA aplikací, nicméně tato varianta není zcela bezpečná a v případě škodlivé aplikace hrozí odcizení přístupových údajů. V praxi lze také narazit na typ *Bearer-only*, který standard nespecifikuje a slouží spíše pro označení klienta jako *resource server*.

3.2.2 OpenID Connect

Jelikož OAuth2 řeší pouze autorizaci uživatele na základní úrovni, aplikace (klient) nemůže koncového uživatele nijak jednoznačně identifikovat. Z tohoto důvodu vznikl standard OpenID Connect (OIDC), který staví na OAuth2 a rozšiřuje tento framework o identifikaci uživatele. OIDC zde specifikuje dva typy komunikace:

- Authorization grant

[39] https://tools.ietf.org/html/rfc6749#section-2.1

- Implicit grant

V případě *Authorization grant* je uživateli vytvořen autorizační token, který předává klientovi. Klient si následně od IdP vyžádá přístupový token, kterým se pak ověřuje vůči ostatním službám. OIDC zde také specifikuje REST endpoint[40], který vrací identitu uživatele.

V případě *Implicit grant* zde již v procesu nefiguruje žádný klient, ale přístupový token je zde předán přímo koncovému uživateli[41]. Token je pak připojen k nezbytným požadavkům pro získání informací. Tato metoda poskytuje pouze zdánlivý pocit bezpečí (škodlivá aplikace se k přístupovému tokenu dokáže dostat, nehledě na více vektorů útoku) a je vhodnější používat spíše authorization grant. OIDC také přesně specifikuje strukturu předávaných dat, důležitý je obzvlášt atribut 'sub', který jasně specifikuje koncového uživatele (email, uživatelské jméno, či jiné).

3.2.3 JWT

Jelikož OAuth2 řeší pouze proces jako takový a neřeší zabezpečení předávaných dat[42], je třeba zajistit integritu a bezpečnost dat přenášených přes dešifrovatelný TLS provoz. RFC 7523[10] i OIDC standard zde specifikují využití JWT pro tyto účely. Princip *JSON Web Tokens*[43] je velmi podobný zašifrovaným XML dokumentům známým ze SAML 2.0, tedy šifrování JSON struktury pomocí certifikátu jedné strany, kde strana druhá může pravost dokumentu ověřit pomocí známého otisku certifikátu. Integritu dat s použitím JWT zabezpečuje *Message Authentication Code* (MAC, autentizační kód zprávy), výsledný algoritmus tedy kombinuje integritu dat i šifrování. Struktura JWT zprávy obsahuje hlavičku[44][11][12], ve které je specifikován typ zprávy (media type) a použitý algoritmus (nejčastěji použitý HS256[45]). Tělo zprávy již obsahuje požadované uživatelské atributy, kde JWT standard detailně popisuje i konkrétní atributy a jejich účel.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
    eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
    TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

Výpis 4: Šifrovaná JWT zpráva

[40] Přístupový bod
[41] Koncový uživatel zde může být prohlížeč (např. SPA aplikace), mobilní aplikace, aj.
[42] V případě šifrování se OAuth2 spoléhá na TLS
[43] RFC 7519, https://tools.ietf.org/html/rfc7519
[44] JOSE (Javascript Object Signing and Encryption)
[45] HMAC s SHA-256

```
{
  "alg": "HS256",
  "typ": "JWT"
}
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Výpis 5: Dešifrovaná JWT zpráva - hlavička následovaná tělem zprávy

Spojením OAuth2, OpenID Connect a JWT nám vzniká velmi silný framework s širokým rozsahem využití, který je podporován ohromnou škálou služeb a poskytovatelů identit. Proto bylo rozhodnuto použít právě tuto kombinaci v implementované službě.

4 Implementace

Pro implementaci enterprise aplikací existuje řada platforem, které mohou být použity jako základ. Velmi používanou platformou jsou jazyky postavené nad JVM, kde i z historických důvodu převažuje jazyk *Java* a jeho webová specifikace *Java EE*. Vzhledem k technologickému rozmachu v oblasti programovacích jazyků a přístupu k řešení škálovatelných problémů, existuje řada alternativ a konkurentů. Velmi oblíbeným přístupem, jakožto alternativou k procedurálním (sekvence příkazů, např. C) a objektově-orientovaným (např. Java) jazykům, jsou jazyky založeny na lambda kalkulu, nazývány jako funkcionální. Díky této oblíbenosti a také díky přehlednějšímu a testovatelnějšímu kódu, pronikají praktiky funkcionálních jazyků i do jazyků nefunkcionálních. Při technologické volbě při novém projektu je tedy velmi důležité přihlédnout i k výhodám odlišných přístupů, přestože větším důvodem pro zvolení konkrétní technologie či platformy by stále měla zůstat zkušenost programátora.

4.1 Volba implementační technologie

Volbu použití technologií může také ovlivnit celá řada faktorů:

- Rozsah aplikace
- Zaměření aplikace
- Robustnost
- Efektivita vývoje
- Dostupnost podpůrných knihoven
- Multiplatformnost

Jestliže je vyvíjená aplikace mířena na velmi malé spektrum uživatelů (resp. velmi malé množství uživatelů), je vhodné zvolit programovací jazyk pouze dle zkušeností programátora. Přestože celá řada vyspělých jazyků a architektur nabízí zajímavé přistupy, v rozmezí malé aplikace může být komplexní platforma kontraproduktivní. Jestliže by využití dané platformy znamenalo razantní zvýšení požadavků na zdroje, je vhodnější zvolit technologie, které umožňují aplikaci provozovat s minimem nákladů (například rozličné skriptovací jazyky a webhostingové technologie - PHP, Python, Node.js, aj.).

Při volbě JVM jako hlavní platformy existuje řada rozdílných přístupu použití. JVM figuruje pouze jako interpret instrukcí Java bytekódu, je tedy možné využít libovolný jazyk, který lze do definovaného bytekódu zkompilovat. Kromě programovacího jazyka Java existuje spousta rozdílných jazyků postavených nad tímto bytekódem, v případě potřeby skriptovacího jazyka je možné využít jazyk *Groovy*, pro potřeby funkcionálních přístupů je možné využít jazyky *Clojure*

(Lisp dialekt) či *Scala*[46]. Existují taktéž jazyky přímo určené jako náhrada jazyka Java (např. *Kotlin*), či pouze interprety existujicích jazyků nad JVM (JRuby, Jython, Nashorn, aj.). Mezi výhody použití JVM jakožto běhové platformy se řadí velká škála použitelných knihoven[47] či JIT[48] a díky komplexnímu *Garbage collectoru* je možné psát vysoce paralelizovatelné aplikace[49] a není zde nutné řešit problémy typu *GIL*[50]. Jestliže se programátor vyvaruje použití funkcí specifických pro konkrétní operační systém (použití nativních knihoven, specifika souborových systémů, aj.), je možné ve velké míře dosáhnout multiplatformnosti.

Jeliže je jedním z požadavků na platformu i efektivita vstupně-výstupních operací, je důležité znát rozličné přístupy v práci s I/O. Obzvláště při práci s velkým počtem síťových operací (problém *C10K*[51]) je standardní přístup, kdy je pro každé spojení delegováno jedno vlákno či proces (blokující I/O) velmi neefektivní. Pro neblokující I/O operace existuje řada přístupů, pravděpodobně nejefektivnějším přístupem je *event loop*, kdy všechny požadavky jsou odbavovány jedním vláknem a v případě potřeby blokující operace (výpočet, práce s I/O) je požadavek daného klienta předán vláknu jinému či pozastaven, dokud nebude blokující operace dokončena. Zmíněný princip adaptují například proxy server *nginx* či prostředí *Node.js*, podobný přístup taktéž využívá většina NIO knihoven[52]. Použitím tohoto principu je možné dosáhnout velmi efektivní práce s klienty a zmíněné technologie jsou využity u velkého množství rozsáhlých webových systému[53][54][55]. Velmi důležitým detailem je také komunikace uvnitř aplikace či mezi řadou aplikací (IPC). Oba problémy je možné efektivně vyřešit využitím fronty zpráv (MQ), při psaní velmi asynchronních algoritmů se jeví vhodnější přístup programovacího jazyka *Erlang*, označovaného jako *Aktorový model*[56]. V aktorovém modelu je každá potřebná součást označena jako objekt, který dokáže přijímat delegované zprávy a vygenerovat potřebnou odpověď. Tento princip dokáže velmi efektivně distribuovat[13] události napříč systémem a zajistit notifikaci potřebných součástí (event-based systém). Pro využití daného principu nad JVM existuje například knihovna *Akka*, či implementace pomocí rozličných MQ systémů (např. RabbitMQ).

Díky praktickým zkušenostem a rozšířenosti platformy Java byla tato platforma zvolena i pro implementaci platformy OpenDVS. Nebyla zvolena čistá verze Java EE, ale pro programátorskou přívětivost a rozšířenost byl zvolen framework *Spring Boot*. Pro uživatelskou část byla zvolena

[46] Kombinace přistupu z jazyků Erlang, Haskell, Scheme, Smalltalk či ML-related jazyků, založeno na silné typové kontrole a funkcionálním přistupu

[47] Vzhledem k interpretaci bytekódu je možné využívat knihovny psané v libovolném jazyce běžícím nad JVM

[48] Just-in-time kompilátor

[49] Od verze Java 7 existuje G1 garbage collector vhodný i pro velké rozsahy operační paměti, vlákna nad JVM jsou mapovány na nativní vlákna operačního systému

[50] *Global interpreter lock* - mutex řešící synchronizaci běžících vláken pro aktuální běh pouze jednoho z nich současně, problém základních interpretů jazyků Python, Ruby, aj.

[51] http://www.kegel.com/c10k.html

[52] V případě použití na operačním systému Linux je zde ve většině případů využita kernel funkctionalita *epoll*

[53] https://www.nginx.com/videos/cloudflare-boosts-performance-and-stability-for-its-millions-of-websites-with-nginx/

[54] https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/

[55] https://www.nginx.com/blog/why-netflix-chose-nginx-as-the-heart-of-its-cdn/

[56] Actor-based concurrency

knihovna React s podpůrnou knihovnou Redux.

4.2 Architektura

Vzhledem k požadavkům na platformu je možné rozdělit funkcionalitu do tří logických operací:

1. Detekce závislostí
2. Získání metadat komponent
3. Analýza zjištěných informací

Při volbě hlavní architektury je velmi důležité definovat datový model a dbát na platformní požadavky. Velkým problémem přílišné fyzické separace komponent, obzvláště v případě microservices, je definování autorizace. V případě, že je autorizace založena na systému ACL[57], je pro minimalizaci duplicit vhodné využívat pouze jednoznačné identifikátory entit pro značení oprávnění. Jestliže chce být programátor věrný konceptu microservices, nabízí se vyseparovat autentizační a autorizační službu do separátní aplikace, kde z důvodu bezpečnosti má aplikace vlastní oddělenou databázi. Při využití OAuth2 je možno ostatní části aplikace označit jako *Resource Server* a pro ověření požadavku lze efektivně využit *Bearer token*. Nyní nám vzniká zásadní kolize s uživatelskou přívětivosti. Pokud autorizační služba, která tedy oznamuje ke kterým identifikátorům má uživatel specifický přístup, poskytuje pouze identifikátory, jak je možno provádět stránkování, řazení a filtraci dat? Z hlediska uživatelské přívětivosti chceme jistě uživateli poskytnout možnost vyhledávání, provádět filtraci oproti specifické hodnotě entit a následný vystup seřadit a stránkovat[58]. Jestliže chceme zároveň minimalizovat přenášená databázová data, jakožto nutnou podmínkou při tvorbě enterprise aplikací, vznikají nám dvě logické varianty, jak daný problém vyřešit. Prvním řešením může být předávání celého seznamu identifikátorů potřebným částem platformy a dané operace (filtrace, řazení, stránkování) budou již prováděny přímo na databázi. Toto řešení má problém v případě velkého množství identifikátorů. Kdyby například platforma nabízela specifikovat oprávnění pro každou detekovanou závislost[59] a jesliže základní Spring Boot aplikace obsahuje 67 závislostí, v případě komplexních aplikací bude počet závislostí vysoký a daná implementace není efektivní. Druhým řešením může být uchovávaní filtračních a řadících dat na autorizační službě. Tento problém částečně řeší definovaný problém, jelikož částem platformy předáváme již seřazené a filtrované identifikátory entit. Zde ale dochází k zbytečné duplikaci dat. Třetím řešením je nejprve provádět filtraci a řazení na částech platformy a před konečným předáním koncovému uživateli provést filtraci dle daného oprávnění. Tento způsob je však zdaleka nejhorším, nelze efektivně zjistit počet odpovídajících

[57] *Access Control List* - pro každou spravovanou entitu jsou definována granulovaná oprávnění, která mohou být přiřazena uživateli či skupině

[58] Pro efektivní využití paměti a nepřetěžování služby

[59] Například pro zobrazení celého seznamu komponent a jejich informací

entit a hrozí, že uživateli bude vrácena prázdný seznam entit pro velký počet požadavků. Proto toto řešení nebylo dříve zmíněno a každý programátor by se měl podobným řešením vyvarovat.

Ze zmíněných důvodů byla zvolena hybridní architektura[60], kdy existuje jádro platformy provádějící autorizaci, které má přímý přístup do databáze spravovaných entit, následné logické operace jsou vyseparovány do oddělených komponent. Celou platformu tedy tvoří:

- Jádro platformy
- Aplikace pro detekci závislostí
- Aplikace pro stahování metadat komponent
- Aplikace pro analýzu zjištěných informací[61]
- Notifikační aplikace (emaily a přidružené integrace)

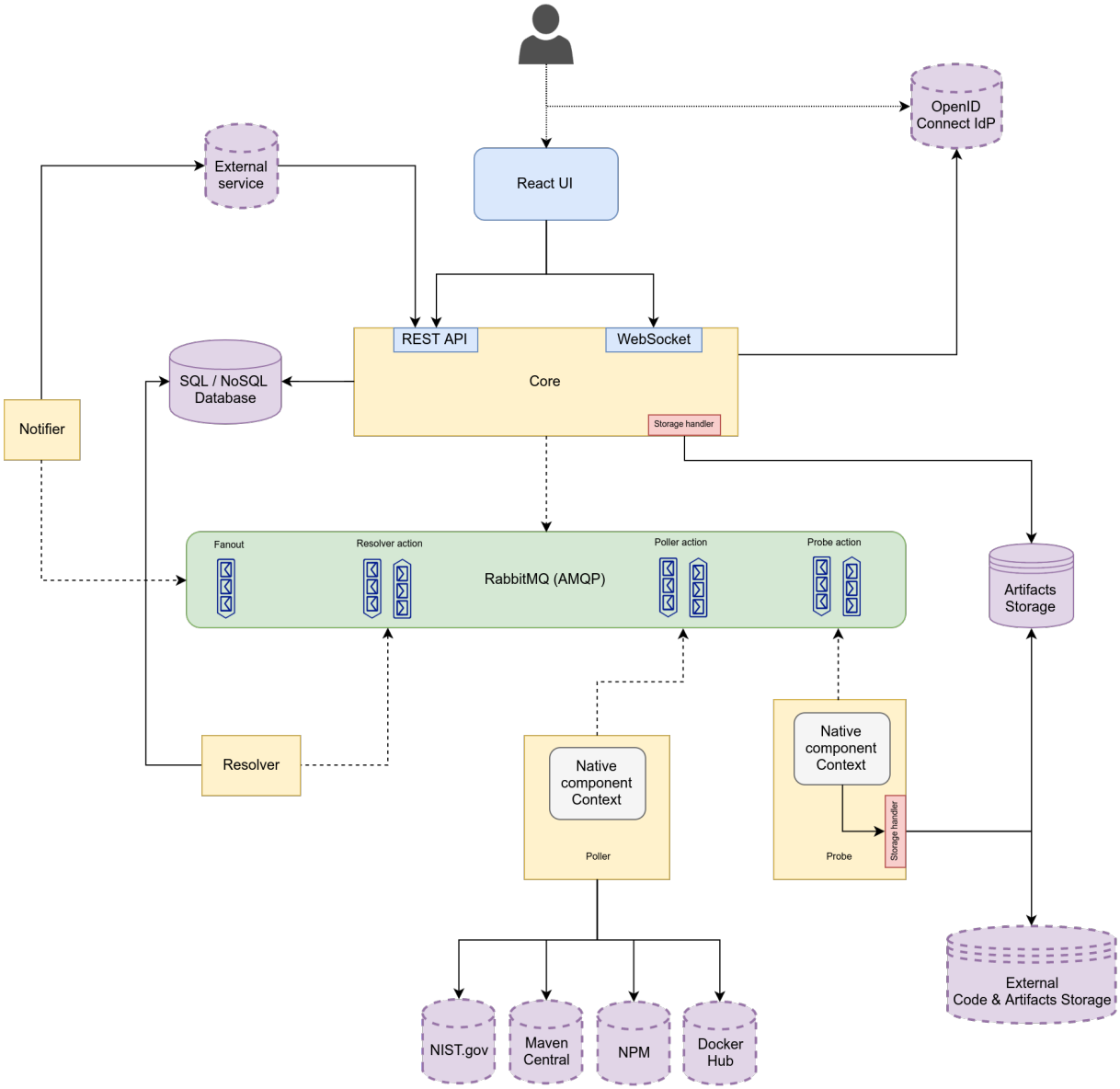
Díky koncipaci platformy jako open-source produkt jsou jednotlivé komponenty označeny anglickými termíny. Detekční sondy jsou ozačeny jako *Probes*, sondy pro získávání informací o komponentách jsou označeny jako *Pollers* a komponenta pro analýzu informací je nazvána *Resolver*.

4.3 Jádro platformy

Jádro platformy poskytuje unifikovaný přistup k aplikačním datům, zároveň na poskytovaná data aplikuje autorizaci pomocí *RBAC*[62], kde pro přístup ke konkrétnímu projektu je vyžadována role *PROJECT_<ID>* či *PROJECT_<ID>_ADMIN*. V prvotní implementaci nebyl brán zřetel na rozdílné práva konkrétních entit, není tedy aktuálně možné efektivně zobrazit seznam všech detekovaných komponent napříč projekty. Komponenta je postavena na frameworku Spring Boot, v době implementace ve verzi 1.4.1. Jelikož tato komponenta slouží taktéž jako zdroj dat frontendové aplikaci, jsou tato data vystavena přes REST API. Přestože byl brán zřetel na správné implementování RESTful přístupu, není zde aplikován *hypermedia-driven* přístup, tedy jednotlivé vazby mezi projekty jsou představeny pomocí objektů a ne pomocí odkazů na konkrétní REST endpoint. Přestože toto omezení nehraje velikou roli při utilizaci API[63], je vhodné zmínit nedodržení přístupu pro upozornění na zažitý omyl o implementaci RESTful služeb.

Při implementaci byla aplikovaná standardní třívrstvá architektura, existuje tedy prezentační vrstva (REST), servisní vrstva (business logika) a vrstva přístupu k datům. V servisní vrstvě je pro komunikaci s prezentační vrstvou implementován návrhový vzor fasáda, kdy je

[60] Kombinace microservices a monolitické architektury
[61] Komponenta má zároveň přístup do stejné databáze jako jádro platformy
[62] Role-based access control
[63] Při aplikaci HATEOAS konceptu (resp. HAL standardu) je do jisté míry možné API autodiscovery, není nutné znát jeho kompletní popis



Obrázek 7: OpenDVS architektura

zamezeno přílišné integraci mezi oddělenými vrstvami. Fasáda zde složí pro unifikaci aplikačních volání, kombinuje tedy například autentizaci, autorizaci a business logiku. Pro snadnější spojení s databazí je zde využité ORM[64] poskytované komponentou *Spring Data*. Výhodou použití dané komponenty je abstrakce přístupu nad konkrétní ORM implmenetaci. Je tedy velmi jednoduché vyměnit použitou SQL databázi za jinou (pomocí frameworku Hibernate), či poměrně snadné zaměnit SQL implementaci za NoSQL (např. MongoDB). V prvotní fázi je důkladně využíváno relací mezi objekty, je tedy využita SQL implementace. Pro nasazení aplikace je doporučena varianta databáze MySQL (např. Percona XtraDB), nicméně díky korektní práci se specifikací

[64] Objektově-relační mapování

JPA[65] je změna za jinou SQL-compliant databázi otázkou správného nastavení konfiguračního souboru. Jelikož dle navrhnuté datové struktury je možné předpokládat velký objem dat, je této problematice věnována sekce v kapitole 6.

Pro potřeby řazení a stránkování požadavků je zde využívána Spring Data struktura *Pageable*, která je automaticky vytvořena při každém požadavku a obsahuje informace o požadované stránce a řadícím mechanismu. Vzhledem k nedostupnosti filtrovacího mechanizmu u použitého frameworku byla vytvořena speciální struktura *Filterable*, která je, podobně jako struktura Pageable, automaticky[66] vytvořena dle vstupních dat uživatele. Jelikož není možné využít existujících abstrakcí nad Spring Data (tak jako v případě Pageable), bylo nutné vytvořit specifikaci SQL požadavku, který korektně obsahuje správné filtrační podmínky. Jelikož modifikace konkrétního požadavku je velmi náchylná na rozličné vektory útoků (např. SQL injection či DOS), bylo zde využito konceptu *DSL* u JPA, kdy jsou potřebné požadavky zadány pomocí predikátů, ze kterých je zformulovaná konečná specifikace daného požadavku. Jelikož tento princip porušuje nezávislost implementaci pro SQL/NoSQL databázi[67], v případě změny databázového přístupu bude nutné provést translační vrstvu mezi JPA specifikacemi a např. *MongoDB Query DSL*. Specifikovaný problém by taktéž mělo jít vyřešit při využití knihovny QueryDSL, která již tyto mechanismy implementuje.

Pro přehlednou definici REST endpointů je využito frameworku *Swagger*, respektive knihovny *SpringFox*, která integruje přístupy frameworku do konceptu frameworku *Spring Boot*. Každý dokumentovaný endpoint stačí pouze označit anotací *@ApiResponse*, čímž docílíme přidáním endpointu do jednotného JSON dokumentu, který obsahuje kompletní definici praktik, jak s daným API pracovat. Výhodou je jistá standardizace u použitých technologií, je tedy možné například importovat existjící Swagger definice to rozličných API managerů.

4.4 Datový model

Základním platformním typem je *Artifact*, kterým může být míněna ať již finální podoba aplikace (např. spustitelný soubor), ale i zdrojové soubory aplikace[68]. Samotný artefakt obsahuje základní informace o analýze (stav analýzy, príslušná detekční akce, seznam komponent) i vlastní detaily (typ - sestavení / zdrojový kód, URI, název a jednoznačná identita v rámci projektu - např. hash Git commitu). Artefakt také vždy patří do jediného projektu, který je definován dynamickým typem (např. Git projekt) a typovými parametry (např. Git clone URL).

Každá detekovaná závislost je uložena jako *ArtifactComponent*, neboli komponenta artefaktu. Ta obsahuje všechny detekované informace o specifické závislosti - skupinu (např. maven), de-

[65] Není zde použita žádná specifická syntaxe pro MySQL, pro potřeby konkrétního SQL dotazu je využito JPQL

[66] Automatického vytvoření je dosáhnuto za pomocí Spring autokonfigurace a korektní implementace rozhraní *HandlerMethodArgumentResolver*

[67] Tento princip již teoreticky porušuje využití JPA, nicméně je možné buďto využít dvojího značení pomocí anotací, či existující vrstvy pro integraci JPA anotací pro NoSQL databáze

[68] Zde dochází mírně ke konfliktu s existujícím názvoslovím, kdy jako artefakt je označována právě zmínená finální podoba aplikace - výstup po sestavení

tekovanou verzi či hash, rozsah (scope, např. compile), odvoditelný identifikátor[69] a stav (např. zastaralá). Každé komponentě také odpovídá *ProbeActionStep*, kde jsou zaznamenány detaily detekce. Je zde také uveden odkaz na rodičovskou komponentu pro zpětné sestavení grafové struktury.

Při kroku identifikace metadat je každá komponenta (zde již nezávislá na jakémkoliv artefaktu) uložena do struktury *Component* a příslušné verze do struktury *ComponentVersion*. V každé komponentě je uložena pouze informace o skupině, názvu (odvoditelný identifikátor u ArtifactComponent) a poslední verzi komponenty. Přestože při vývoji aplikací se lze setkat např. s přístupem *stable-oldstabe*, kdy je udržováno několik oddělených, stabilní verzí a jiná verze než poslední může být stále aktuální, není tento přístup podporován balíčkovacími systémy (je dostupná pouze nejnovější verze). Pro jisté umožnění zmíněného přístupu je pro projekt definována hodnota *majorVersionOffset*, kdy je komponenta, která byla vydána maximálně *majorVersionOffset* dnů zpátky a zároveň její major verze není shodná s verzí aktuální nejnovější verze (viz. sémantické verzování), označena za aktuální a není s ní pracováno jako se zastaralou. Verze komponenty uchovává potřebné detekované informace, zejména zdroj informací, samotnou verzi, hash (je-li k dispozici), datum vydání a datum synchronizace metadat. Následné propojení detekovaných komponent u artefaktu a detekovaných metadat je prováděno v analyzační aplikaci, zde již není třeba mít k dispozici žádné modely.

Veškeré jednoznačné identifikátory v rámci entit jsou automaticky generovány aplikací pomocí algoritmu UUID[70], není tedy využit *Identity / Auto Increment* přístup na straně databáze. Zvolený přístup jednak nepokládá žádné požadavky na použitou databázi, taktéž je možná znovupoužitelnost smazaných ID. Nevýhodou je samožřejmě větší velikost ukládaného identifikátoru (36 znaků u UTF-16 znamená alespoň 72 bytů oproti 8 bytům pro číslo u 64bit architektury), není nicméně snadné odhadnout existující identifikátory[71].

Mezi všemi částmi aplikace jsou využity *one-to many* vazby (mimo provázanost detekovaných komponent a metadat), jsou zde tedy definovány cizí klíče (foreigh keys) a integrita dat je kontrolována již na úrovni databázového systému.

4.5 JSON Serializace

Velkým problémem při serializaci dat do libovolného formátu jsou oboustranné vazby. Jesliže máme 1-N vazbu ze třídy A do třídy B, existuje nám i opačná vazba ze třídy B do A. Jesliže jsou tyto vazby nastavovány automaticky (např. za pomocí ORM frameworku), vzniká nám zde velké riziko, že při serializaci nám dojde k zacyklení. Pokud daná serializační knihovna dokáže počítat již serializované/serializující objekty a zároveň výstupní formát dokáže tyto informace zaznamenat, není třeba tento problém řešit. V případě serializace do formátu JSON a při použití

[69] Identifikátor je jednoznačný pro konkrétní skupinu - například *maven:me.raska.opendvs:opendvs-core*

[70] 36-znakový řetězec (32 náhodných znaků) ve formátu *xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*

[71] Přestože by bylo možné se spolehnout na náhodnost a dostatečnou kryptografickou složitost využitého algoritmu, je každá práce s entitou vázána na konkrétní oprávnění a přístup je explicitně ověřován. Není zde tedy využito chybného principu *Security through obscurity*

Java knihovny Jackson zde takováto funkcionalita neexistuje, je důležité klást důraz na správnou definici serializačních pravidel.

Pokud je publikovaná struktura *flat*[72], je možné každou vazbu pouze označit anotací *@JsonIgnore*, kdy při serializaci bude tato vazba ignorována, zároveň při přístupu z dané aplikace zde nedojde ke změně chování. Pokud by byl aplikován přístup HATEOAS, byly by taktéž všechny vazby pravděpodobně nahrazeny hypertextovými odkazy a nebylo by nutné řešit tyto problémy.

Jesliže je nicméně vyžadována vícevrstvá struktura (např. aby byl při získání artefaktu vrácen i identifikátor), je nutné využít alespoň anotaci *@JsonIdentityInfo*, která místo objektu vráti pouze hodnotu jednoznačného identifikátoru objektu[73]. Tento přístup je efektivní pouze v případě, že není třeba publikovat i vedlejší informace, které by usnadnily uživatelskou přívětivost (např. jméno projektu, aby nebylo nutné pro zobrazení dalších informací vytvářet další REST požadavek).

Pokud je vyžadovaná komplexní struktura, je možnost využít anotaci *@JsonIgnoreProperty*, kde je možné definovat, jaké hodnoty z přiřazené struktury se mají zobrazit. Je možné tedy ignorovat zpětnou vazbu, či jiné vztahy přidružené třídy. Velkou nevýhodou je chybějící anotace (např. *@JsonAllowProperty*), která by měla za cíl povolit pouze specifické hodnoty přidružené třídy, jelikož při *@JsonIgnoreProprety* je nutné explicitně zakázat nové vazby.

Jelikož Jackson knihovna neumožňuje v základu počítání referencí (resp. objektů, které zpracovává), dochází ke zbytečné duplikaci dat (stejné objekty se mohou vyskytovat vícekrát v daném dokumentu), což vzhledem k JSON standardu není překvapivé. Za cíl vyřešit tento problém (a současně i "nekonečnou serializaci") má specifikace JSOG[74], kdy je každé entitě přiřazena vlastnost *@id*, obsahující unikátní identifikátor v rámci serializovaného dokumentu. Při serializaci je udržováno mapování *objekt -> @id* a v případě stejného objektu je jeho stuktura reprezentována pouze pomocí objektu s hodnotou *@ref*. Přestože tato specifikace není hojně rozšířená, existuje Jackson generátor *JSOGGenerator*[75], poskytující nezbytnou funkctionalitu a autokonfiguraci. Pro klientskou stranu existuje JavaScript knihovna, která dokáže generovanou strukturu dekódovat. Proto je také této knihovny v projektu OpenDVS využíváno.

4.6 Messaging

Pro propojení jednotlivých komponent platformy se nabízí řada technologií. Pro autokonfiguraci komponent dle konfiguračních parametrů lze využít například *Spring Cloud Config*, alternativou může být například *Netflix Eureka*. Úkolem podobných config serverů je dodat počáteční konfiguraci dané aplikaci, taktéž dokáže informovat připojené klienty o vzniklých změnách či připojených klientech a jejich rolích. Důležitější zde nicméně je samotná komunikace mezi jednotlivými prvky systému. Jak již bylo zmíněno, v případě, že je třeba poslat informaci a není

[72] Flat struktura neobsahuje žádné zanořené elementy
[73] Jedná se zde sice o flat strukturu, ale s informací o navázaných relacích
[74] JavaScript Object Graph
[75] https://github.com/jsog/jsog-jackson

nutné mít zajištěné zpracování, přímá komunikace (např. přes REST) nezpůsobuje problémy. Jestliže ale je nutná garance zpracování zprávy, je třeba aplikovat příslušné architektonické vzory.

Jeliže se bavíme v kontextu enterprise aplikací, disciplína EAI (Enterprise Application Integration) se věnuje této rozsáhlé problematice. Nejčastěji aplikovaným vzorem zde jsou *fronty zpráv* (message queue, MQ), kdy při použití *dead letter queue* je umožněno uchovat i neúspěšně zpracované zprávy (implementační chyba, integrační chyba, timeout, aj.) a je umožněno tuto zprávu opětovně zpracovat po vyřešení vzniklých potíží. V platformě OpenDVS je MQ využíváno hojně. Jelikož architektura platformy je navržena pro efektivní škálování, jsou jednotlivé komponenty na sobě co nejvíce nezávislé a veškeré dlouhotrvající akce nejsou prováděny jádrem platformy. Jelikož veškeré detekční akce jsou prováděny v aplikacích označených jako *dělnící* (workers), nabízí se využití právě MQ společně s DLX pro zachování neúspěšně zpracovaných zpráv a to s využitím fronty typu *direct*. Úkolem dělníka je přijmout zprávu, provést rozličné operace a vygenerovat výstup. Framework Spring Boot umožňuje snadné využití knihovny *Spring AMQP*, která obsahuje implementaci s existujícími MQ systémy právě za pomoci protokolu AMQP[76]. Zde je využito automatické konfigurace potřebných parametrů na straně *Message brokera*[77] za pomoci definování aplikačního kódu (pomocí Bean definic). Pro produkční nasazení je doporučen broker *RabbitMQ*, nicméně jakýkoliv broker podporující protokol AMQP 0.9.1 je podporován. Pro dosažení efektivního horizontálního škálování jádra platformy je také definována fronta zpráv typu *fanout*, která zprávu předá všem připojeným klientům. V případě, že tedy dojde ke změně stavu, o které by měl být koncový uživatel notifikován, je tato informace předána do této fronty a je automaticky rozdistribuována všem připojeným aplikacím a může být dále předána uživatelům pomocí technologie WebSocket[78], či aplikovány rozličné notifikační pravidla[79].

4.7 Sondy

Pro analýzu konkrétních artefaktů je definovaná oddělená komponenta, označena jako *Probe worker*[80]. Samotná logika komponenty je velmi jednoduchá, při spuštění aplikace jsou pomocí reflexe detekovány všechny implementace rozhraní *NativeProbe*, které definuje dvě akce - detekci a extrakci. Detekce má za cíl v předaném seznamu cest detekovat komponenty daného typu (např. Maven), extrakce má za cíl vygenerovat cesty do daného seznamu (např. archivy). Samotná extrace můsí být prováděna do oddělených adresářů, doporučeny jsou dočasné adresáře s náhodným názvem, jelikož jsou poté automaticky mazány. Komponenta mimo detekci sond provádí i detekci implementaci rozhraní *ArtifactSource*, které definuje získání konkrétního arte-

[76]Správné označení by bylo protokol AMQP 0.8 až AMQP 0.9.1. Vzhledem k existující specifikaci AMQP 1.0, která popisuje diametrálně odlišný protokol, je důležité zmínit tento rozdíl.

[77]Message broker označuje MQ systém poskytující potřebnou funkcionalitu

[78]Klienti připojení pomocí technologie WebSocket se připojují pouze na jeden uzel aplikačního clusteru, je tedy nutné zprávu rozdistribuovat na všechny uzly

[79]Zde je třeba rozlišit, zda se jedná o notifikaci připojených klientů (pomocí fanout typu), či se jedná o notifikaci prováděnou pouze z jednoho aplikačního clusteru

[80]Překlad do češtiny by mohl být pravděpodobně *sondovací dělník*

faktu (např. Git či souborové úložiště), detekované implementace jsou následně instanciovány a uloženy do příslušných kolekcí (množina a mapa). Samotná detekce je definována jednoduchou posloupností kroků:

1. Získej artefakt
2. Dokud existují extrahované složky
   (a) Pro každou sondu detekuj komponenty
   (b) Pro každou sondu extrahuj komponenty
3. Smaž dočasné soubory a složky
4. Odešli výsledek detekce do příslušné fronty zpráv

Přestože zmíněný přístup evokuje použití architektonického vzoru *Pipes and Filters*, nebyl zde tento přístup aplikován, taktéž nebylo využito vícekrokového použití fronty zpráv, například pro opakování pouze konkrétního detekčního kroku.

Jelikož jsou jednotlivé sondy načítány pomocí reflexe, stačí, aby konkrétní implementace byla dostupná v classpath jednotlivého dělníka. Při využití spustitelného JAR souboru je možné definovat správný *loader.path* parametr s cestou ke složce implementací, není tedy nutné pro přidání implementace kompilovat celého dělníka. Základní sada sond je pomocí systému Maven přibalena přímo k dělníkovi, v budoucnu by tedy bylo vhodné vyseparovat i tyto knihovny jako oddělené, například pro efektivnější opravování chyb.

```
public interface NativeProbe {

    List<ProbeActionStep> extract(Artifact artifact, List<ArtifactComponent>
        extractedComponents,
            ProbingContext context);

    List<ProbeActionStep> detectComponents(Artifact artifact, List<
        ArtifactComponent> extractedComponents,
            ProbingContext context);
}
```

Výpis 6: Sondovací rozhraní NativeProbe

4.8 Detekce verzí závislosti

Pro získání metadat o konkrétní komponentě je využita obdobná architektura, jako u sond. Dělníkem zde je komponenta *poller-worker*, která na základě vstupu provede potřebnou operaci

a vygeneruje výstup. I zde je pro modularitu a snadné rozšiřování využito reflexe a rozhraní *NativePoller*. Pro snadnější použití je zde využito funkcionální rozhraní *Consumer*, ve kterém je předán odkaz na funkci pro odeslání zprávy do MQ.

Při implementaci dané funkcionality nám vznikají poměrně zásadní problémy:

- Jak získat metadata specifické komponenty?
- Jak získat metadata všech komponent?
- Jak pravidelně aktualizovat metadata?

První dva problémy jsou z velké míry specifické pro konkrétní skupinu komponent. V případě Maven repozitářů je cesta komponenty odvozena z identifikátoru, například komponenta *spring-boot-starter-web* ze skupiny *org.springframework.boot* má metadata uložena na adrese *http://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-starter-web/maven-metadata.xml*. Problémem nicméně je, jestliže potřebujeme informace o všech komponentách, jelikož takováto databáze nemusí být dostupná. Z tohoto důvodu byla vymyšlena dvojí logika získávání metadat:

1. Jesliže repozitář poskytuje databázi komponent, je pravidelně získávána (filtr *<skupina>:**)
2. Jesliže repozitář neposkytuje kompletní databázi, jsou metadata stahována na požádání (filtr *<skupina>:<odvoditelné ID>*)

Dle zmíněného rozdělení lze třetí otázku rozděli do dvou skupin, aktualizace databáze komponent a aktualizace konkrétních komponent. Jestliže je poskytována databáze všech komponent repozitáře, je vhodné implementovat pouze zpravování wildcard filtru. Pokud takováto databáze poskytována není, je vhodné neimplementovat rekurzivní procházení repozitáře (např. pomocí directory indexu) a wildcard filtry ignorovat. Pokud jsou tato doporučení dodržena, je možné pravidelně aktualizovat metadata o známých komponentách, nejlépe dle fixně specifikovaného intervalu (např. každý den ve 3 ráno). Pro každou skupinu komponent bude odesláno N zpráv, jedna wildcard a jedna pro každou uloženou komponentu v databázi. Tímto lze dosáhnout aktualizace metadat a tím splněné podmínky pro kontinuálně udržovaný stav aplikace.

Jelikož častým stahování metadat (či rekurzivním stahováním adresářů) je zatěžována kapacita linky poskytovatele repozitáře i provozovatele systému OpenDVS, je vhodné nastavit jistou retenční politiku. Doporučená doba je 24 hodin, ve které je při požadavku o nová metadata komponent zabráněno stahování. Tato doba je konfigurovatelná, je tedy na uvážení provozovatele, na jakou hodnotu bude nastavena. Za předpokladu korektního nastavení lze tedy provozovat tuto službu bez zbytečného stahování redundantních informací.

```
public interface NativePoller {

    String getId();

    void process(PollerAction action, Consumer<PollerAction> callback);
}
```

Výpis 7: Rozhraní NativePoller

4.9 Detekce zranitelností

Pro klasifikaci komponenty jako zranitelné je možné využít volně dostupné zdroje potřebných informací. Ze zmíněného portálu *National Vulnerability Database* není možné efektivně detekovat, zda je daná komponenta skutečně zranitelná[81], byla tedy implementace vytvořena velmi obecně a bude následně rozšiřována. Jakožto zdroj informací zde byla využita databáze CVE poskytována daným portálem, kdy ověření zranitelností je prováděno oproti tagu *vuln:product*, který musí obsahovat jak část identifikátoru, tak i verzi. Zde aktuálně není realizována kontinuální analýza (v případě nalezení nové zranitelnosti) a to z důvodu přílišné náročnosti na zdroje (full-text vyhledávání). Do budoucna by byla vhodné využívat indexačních nástrojů (například některých NoSQL databází), které vytváří index na vkládaných datech, či důsledná dekompozice získávaných dat o zranitelnostech.

4.10 Reaktivní frontend

Pro tvorbu klientské části byla zvolena JavaScriptová knihovna React, pro zajištění uniformního stavu aplikace byla využita knihovna Redux, je tedy teoreticky možné stav aplikace uložit a případně načíst. Pro přehlednou vizuálnost tvořeného UI byla zvolena knihovna Material-UI[82], která implementuje komponenty požadované dle zásad Google Material Design. Výhodou využití zmíněných knihoven je kupříkladu reaktivnost celé aplikace, je tedy možné efektivně notifikovat uživatele o vzniklé akci[83]. Pro komunikaci s jádrem platformy je využita struktura JSON a definované API, byla zde použitá základní knihovna pro práci s HTTP požadavky (*fetch*, náhrada staršího *XMLHttpRequest*), v budoucnu by mohla proběhnout změna na knihovnu *axios*, v případě, že by bylo nutné rušit existující HTTP požadavky (např. autocomplete funkcionalita).

Problémem při implementaci SPA je výběr autentizačního mechanismu a jeho implementace na backendu systému. Jelikož nejenom v případě aplikací je při načtení stránky ztracen kontext a stav aplikace, je důležité vhodně implementovat přihlašovací mechanismus. Jesliže je zvolen

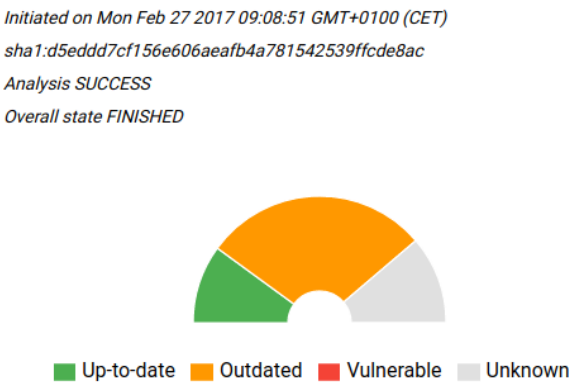
[81] Absence jednotného jednoznačného identifikátoru komponenty

[82] http://www.material-ui.com/

[83] Jelikož má aplikace perzistenně navázáné WebSockets spojení, je notifikace možná i při změně jednotlivých stránek (views)

OpenID Connect s confidential typem, jako v případě platformy OpenDVS, nabízí se periodicky provádět dotazy na backend systému pro obnovení session a bearer tokenů a samotnou autentizaci provádět odděleném okně.

Jelikož je nutné být na straně klientské aplikace notifikován o úspěšném přihlášení, je třeba znát omezení prohlížečů. Jestliže je ze stránky vyvoláno nové okno, je možné dostávat notifikace pouze v případě, že okno patří pod stejnou doménu. Implementace je v tomto ohledu poměrně triviální, uživateli je zobrazena stránka na dané doméně, která provede přesměrování na zvolené IdP. Po úspěšném přihlášení je uživatel přesměrován zpátky na předem zvolenou stránku platformy, kde lze již pomocí JavaScriptu stránku zavřít a tím dojde v SPA aplikaci k upozornění na zavření okna. Zde již můžeme ověřit, je-li uživatel opravdu přihlášen (uživatel může manuálně stránku zavřít i před přihlášením), a pokračovat v požadované akci. Je také velmi vhodné si uchovávat v globálním stavu aplikace informaci, zda přihlašování aktuálně probíha, a následně blokovat všechny generované AJAX požadavky, dokud není proces autentizace úspěšně dokončen.



Obrázek 8: Informace o analýze projektu

Implementovaný frontend v rámci diplomové práce nabízí pouze minimální funkctionalitu, běžnému uživateli nabídne vytvoření vlastního projektu s definovanými parametry, zároveň umožňuje nad tímto projektem provádět analýzy. V rámci nabídnutých informací umožňuje zobrazení detekovaných komponent a jejich stav,zároveň umožňuje nad těmito daty vyhledávat, filtrovat a taktéž zobrazit ucelenou statistiku. Pro grafickou vizualizaci dat[84] je využita knihovna *Recharts*, přes kterou je možné tato data zobrazit jako hierarchickou strukturu, či jako neuspořádanou množinu.

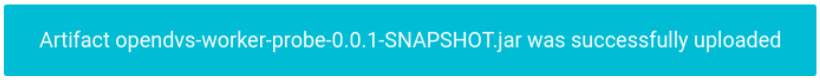
Obrázek 9: Informace o verzích komponenty

Jelikož je vzhledem ke statistice a grafovému zobrazení nutné celá data načíst do paměti prohlížeče, je v případě velkého projektu dat odezva celé aplikace degradována.

[84]Kolektovaná data de-facto reprezentují orientovaný graf, je tedy vhodné ho takto zobrazit

4.11 WebSocket

Pro kontinuální notifikaci uživatele o vzniklých událostech je použita již zmíněná technologie WebSocket. Jelikož tato technologie pouze definuje komunikační kanál a již ne protokol výměny zprávy, byl v platformě využit protokol *STOMP*, který definuje strukturu předávaných dat a to v textovém formátu. Na straně klienta je využívána knihovna *stomp.js*, kdy pro každou přijatou zprávu je uživateli zobrazen notifikační panel typu *Snackbar* s danou informací. V případě, že přijatá zpráva se týká aktuálně otevřené entity (projekt, artefakt), jsou zobrazené informace adekvátně upraveny (např. zobrazení detekovaných komponent po úspěšné detekci).



Obrázek 10: Notifikace skrz Snackbar

Na straně jádra platformy je využita integrace frameworku Spring, konkrétně pomocí modulu *spring-boot-starter-websocket*. Zde je uživateli odeslána jakákoliv notifikace, která odpovídá platným oprávněním. Přestože je možné využít i RabbitMQ broker pro připojení klientů přes STOMP protokol na technologii WebSocket, z důvodu bezpečnosti by bylo nutné dynamicky generovat množství front zpráv a také předávat autorizační informace pro připojené klienty, nebylo toto řešení využito.

4.12 Maven

Detekce Maven závislostí sebou nese poměrně problematickou situaci. Díky možné hierarchické struktuře za pomocí *multi-module* konfigurace nemusí být vždy dostupná všechna potřebná metadata. Prvním krokem je detekce všech projektových souborů *pom.xml*, pro které je následně provedena analýza. Zde se již setkáváme s prvním problémem, tedy detekované projekty nemáme hierarchicky uspořádané a není zde tedy jasná vazba. V aktuální verzi platformy OpenDVS nebyl tento problém efektivně vyřešen, do budoucna je tedy plánován refaktoring.

Pro zpracování POM souborů je možné využít libovolnou knihovnu pro práci s formátem XML. Chceme-li efektivněji pracovat s danou strukturou, nabízí se využít třídu *MavenXpp3Reader* z knihovny *org.apache.maven:maven-model*, která převádí daný soubor na definovaný Java *Model*[85]. Jelikož ale mohou být vlastnosti projektu děděny z rodiče, zadány pomocí proměnných či v případě využití BOM[86] nemusí být specifikovány ani konkrétní verze závislostí, nastává zde důležitý problém. Jelikož načtení pomocí *MavenXpp3Reader* netvoří kompletní strukturu projektu, ani neprovádí sestavení tzv. *Effective POM*, tedy POM souboru obsahujícího všechny přidružené vlastnosti bez použití proměnných, není možné analyzovat kompletní metadata. Problému sestavení *Effective POM* bylo věnováno velké množství času, bohužel v rámci rozumného rozsahu nebylo zjištěno efektivní řešení. Využitou alternativou je spouštění Maven

[85] Třída *org.apache.maven.model.Model*
[86] Bill of materials

pluginů pomocí knihovny *Maven Invoker*, kdy je možné spouštět Maven příkazy z Java kódu. Zde je možné využít tvorbu závislostního stromu za pomoci plugin goalu *dependency:tree* s volitelnou strukturou, který nicméně vyhodí vyjímku v případě neznámé závislosti[87]. Pro potřebu setavení *Effective POM* je možné využít plugin goal *help:effective-pom*, který do souboru (cesta předána parametrem) vygeneruje potřebný XML soubor. Přestože tento soubor nehavaruje při neznámé závislosti, výstupem je pouhá informace o závislostech, tedy bez hierarchické struktury.

Implementace detekce se nakonec rozkládá do tří kroku, které na sebe navazují v případě nezdaru kroku předchozího:

1. Snaha o sestavení závislostního stromu (formát TGF[88])
2. Snaha o sestavení *Effective POM*
3. Načtení čistého POM a nahrazení známých proměnných hodnotou

Pro získání Maven závislostí byl vytvořen poller pro *Maven Central* (repo.maven.apache.org/maven2/) s předem definovaným chováním. Jelikož při využití repozitářových systému (např. *Sonatype Nexus* či *JFrog Artifactory*) existuje jistá databáze metadat (pro Maven Central existuje *Lucene index*[89]), tyto informace nejsou vždy kompletní a slouží spíše pro vyhledávací potřeby. Z tohoto důvodu je implementace provedena detekcí konkrétní komponenty než stahováním celé databáze.

4.13 Npm

Analýza npm souborů je velmi jednoduchá. Celá struktura je definována souborem *package.json*, kde již z přípony je patrná JSON struktura. Celá struktura je reprezentována JavaScript objektem (slovník / mapa, dle terminologie), kde nejzajímavějšími atributy jsou *dependencies* a *devDependencies*, závislosti lze teda rozdělit pro rozsah využití *compile* a *test*. Výhodou balíčkovacího systému *npm* je neexistence multi-module struktury a přestože lze využít proměnné, není to ve velké míře využito. Jelikož ze zmíněných výhod není třeba řešit obdobný problém problému sestaveni *Effective POM* u platformy Maven, stačí sestavit pouze strom závislostí. Pro danou funkctionalitu lze využít řadu knihoven (např. knihovna *madge*[90]), nebyla žádná v projektu využita a aktuálně je detekována pouze "placatá"struktura závislostí.

4.14 Java knihovny

Jelikož Java knihovna (*JAR*, *WAR*, *EAR*) je interně reprezentována jakožto ZIP archiv, je vhodné s ním takto nakládat. Pro detekci daného archivu je vhodné kontrolovat typ souboru,

[87] Zde by mohlo být řešením využít implementaci *M2Eclipse*, která i v případě neznámé závislosti vygeneruje alespoň částečný strom

[88] *Trivial Graph Format*, velmi jednoduchá struktura na programové zpracování

[89] https://lucene.apache.org/core/

[90] https://github.com/pahen/madge

zda odpovídá známému názvu a následně daný soubor zpracovat. Pro zamezení duplicitní logiky při detekci ZIP archivu a Java knihoven byla tato sonda sloučena. Pro implementaci extrakce byla využita ZIP knihovna obsažena ve standardní knihovně jazyka Java (*java.util.zip*), kdy je pro každý detekovaný archiv vytvořen dočasný adresář s náhodným názvem. Pro zamezení jistým bezpečnostním hrozbám je každý název extrahovaného souboru název sanitizován (odebrání dvou následných teček), aby nedocházelo k přepsání souborů v jiných adresářových strukturách.

Jelikož v případě nalezeného archivu není možné jednoznačně zjistit název či verzi komponenty, je vytvořen hash daného souboru a kontrola je prováděna oproti známým hashům[91]. Pro správnou detekci všech archivů je procházena celá adresářová struktura a každý soubor je ověřován oproti následujícím typům:

- *application/x-java-archive*
- *application/x-webarchive*
- *application/java-archive*
- *application/zip*

[91] Přestože Maven Central poskytuje hash Java knihoven, vzhledem k použitému přístupu (on-demand dotazování) nemusí být knihovna správně ověřena v případě malé znalostní báze

5 Konfigurační management

Pro sestavení celé platformy je využit nástroj Maven. Přestože je primárně určen pro sestavování Java aplikací, díky možnosti rozšíření lze využít i pro sestavování aplikací rozličných programovacích jazyků. Struktura projektu využívá modulární konfiguraci, každá komponenta je tedy vlastním projektem, pro kterou existuje nadřažený rodičovský projekt. Vzhledem ke strukturování Spring Boot aplikací (BOM konfigurace), je rodičovskému projektu nastaven rodič Spring Boot. Díky tomuto nastavení probíhá automatická konfigurace verzí Spring Boot komponent, jakožto i sestavování spouštěných JAR souborů. Menší nevýhodou této konfigurace je nutnost stažení závislostí potřebných pro Spring Boot aplikace a to i v případě sestavování nesouvisející komponenty.

Pro sestavení React aplikace je využíván nástroj *npm* společně s nástrojem *webpack*, kdy sestavení aplikace vygeneruje statické soubory s výslednou aplikací. Při sestavování React aplikace jsou taktéž prováděny rozdílné optimalizační techniky, výsledný kód je minimalizován a vygenerován do jediného souboru. Pro integraci s nástrojem Maven je využit plugin *com.github.eirslett: frontend-maven-plugin*, který taktéž zajistí instalaci podpůrných knihoven a aplikací potřebných pro úspěšné sestavení.

Pro automatickou konfiguraci komponent lze použít již zmíněný *Sprint Cloud Config* či *Netflix Eureka*. Jelikož neexistuje přímá komunikace mezi jednotlivými komponentami platformy, ale komunikace probíhá přes MQ broker, je rozdíl konfigurace mezi prostředími jednodušší. Přestože je nutné nastavit manuálně cesty k MQ a databázi, nebyl konfigurační server využit při vývoji platformy. V případě potřeby nasazení je velmi snadné podporu konfiguračního serveru nastavit.

5.1 Databázové migrace

Při dlouhodobém vývoji aplikace může docházet ke změnám datového modelu. Jestliže se spoléháme na relace v databázi, vyžadujeme nastavení nově vytvořených schémat či uložení počátečních dat, namísto vlastní implementace dané logiky je vhodné využít již existujících nástrojů. Pro zmíněný problém existují dvě knihovny, které jsou oficiálně podporované ve Spring Boot aplikacích:

- Liquibase
- Flyway

Princip zmíněných knihoven je velmi jednoduchý. Pro každou spravovanou databázi je uložen identifikátor verze a v případě neshody verze se spouštěnou aplikací jsou provedeny potřebné migrace. Jelikož automatická migrace databázového schématu by mohla způsobit problémy běžící aplikace, zmíněné nástroje operují dle striktně zadaných pravidel. Pro každou databázovou verzi

je definován soubor pravidel, které které budou při migraci provedeny. Tyto pravidla je možné definovat manuálně či je možné pravidla generovat automaticky s následnou manuální revizí. Ve vyvíjené platformě je využívána knihovna Liquibase, převážně i z důvodu většího komerčního nasazení.

5.2 Continuous Integration

Při distribuci zdrojového kódu je velmi důležitou součástí návod, jak danou aplikaci sestavit. Pokud aplikace vyžaduje libovolné závislosti, je třeba tyto informace sepsat a distribuovat se samotným zdrojovým kódem. V rámci OSS aplikací je rozšířeným značkovacím jazykem *Markdown*, kdy za vstupní bod pro dokumentaci aplikace je považován soubor *README* (při použití jazyka Markdown je tímto souborem *README.md*).

V souvislosti se sestavením zdrojového kódu je velmi prospěšná metodika *Continuous Integration* (CI), neboli *Průběžná integrace*. Principem CI je velmi časté zahrnování změn do hlavího zdrojového kódu, obzvláště v rámcí extrémního programování. Cílem častých změn je efektivnější rozdělení změn a snadnější detekce implementačních chyb. Pro dosažení detekce implementačních chyb je nutné pravidelné spouštění jednotkových a integračních testů, nejlépe po každé změně. V případě komunikace s externími systémy je také časté pravidelné sestavování (např. každý den) pro detekci změn na vzdáleném API. V případě práce s verzovacím souborem Git je častá implementace CI pomocí následujících pravidel:

- Každá změna je publikována do vlastní větve
- Pro každou publikovanou změnu jsou spuštěné sestavení včetně testů
- V případě chyby při testování je sestavení označeno za neplatné
- Pro zahrnutí změny do hlavní větve je nutné úspěšné sestavení

Při dodržení daných pravidel je možné detekovat implementační chyby, pokud je daná funkcionalita pokryta testy. Přestože je možné detekovat chybný programovací styl uživatele (např. pomocí *Checkstyle*), nelze vždy efektivně detekovat logické chyby a je vhodné mít zahrnování změn do hlavní větve jako manuální krok vyžadující schválení.

Ze známých nástrojů pro CI je vhodné zmínit nástroj *Jenkins*, který je velmi často využíván, obzvláště díky jeho dlouhé existenci a možnosti rozšiřování. Novější deviací oproti zaběhnutému standardu je definování CI pravidel přímo jako součást zdrojového kódu a využívání kontejnerů pro sestavení aplikace, čehož je možné docílit díky roustoucímu výkonu hardware. Sestavení v rámci nově vytvořeného kontejneru je vhodné jednak z důvodu bezpečnosti, ale hlavním cílem je sestavení aplikace v čistém systému bez možných artefaktů vzniklých předchozím sestavením (např. stažené závislosti či dočasné soubory). Zde je průkopníkem platforma *GitHub* a její systém *Travis CI*, kdy definice sestavení, včetně potřebných závislostí na operační systém, je definována

souborem *.travis.yml* uloženým spolu se zdrojovým kódem. OSS alternativou je platforma *GitLab* a její systém *GitLab CI*, které poskytují totožné možnosti. Na zmíněnou poptávku ve změně CI nástrojů reaguje i nová verze nástroje Jenkins, kdy jsou rozdíly mezi nástroji mnohem menší.

```
language: java
jdk:  oraclejdk8
sudo: false
script:
  - mvn clean install
```

Výpis 8: Ukázka konfigurace nástroje Travis CI

V rámci vývoje platformy OpenDVS byly sepsány i předpisy pro zmíněné CI nástroje platforem GitHub a GitLab, jakožto i dokumentace platformy v jazyce Markdown.

6 Nasazení

Při poskytování aplikace je velmi vhodné distribuovat i instrukce k nasazení aplikace. Velkou výhodou je poskytuntí referenční architektury nasazení (deployment architecture), popisující jednotlivé prvky pro platformu v daném prostředí, včetně komunikačních informací. Obzvláště při nasazení aplikace v microservices architektuře je nutné poskytnout alespoň popis jednotlivých komponent. Jelikož každá instalace dané aplikace může vyžadovat rozdílnou konfiguraci, síťové nastavení či úroveň zabezpečení a garance dostupnosti, pouze jedna referenční architektura nemusí dostačovat. Pro případ bezpečnosti je nutné vydefinovat základní požadavky na komunikaci mezi prvky, požadavky na externí systémy (např. distribuované/centrální úložiště) a výkonnostní požadavky na zdroje.

Obecným rozdělením každé architektury nasazení může být:

1. Nasazení pro vysokou dostupnost
2. Nasazení pro vysoký výkon
3. Základní nasazení bez zmíněných požadavků

V případě vysoké dostupnosti je nutné eliminovat všechny SPOF[92] prvky, je tedy nutné nejméně zduplikovat všechny dostupné komponenty. Tato duplikace vyžaduje nejenom prvky samotné aplikace (tedy jednotlivé aplikační komponenty), ale i veškeré závislé zdroje - úložiště, síťové prvky, fyzický hardware. Pro efektivní zajištění síťové dostupnosti (za předpokladu vysoce dostupných síťových prvků) je možné využít jistou míru rozkládání zátěže (např. DNS loadbalancing), nicméně jako nejefektivnější se jeví využití protokolu VRRP[93], či proprietárního protokolu HSRP[94], které umožní využití tzv. *plovoucí IP adresy*, která bude automaticky přiřazena přávě jednomu zařízení v dané síti.

6.1 Nástroje pro automatizované nasazení

Důležitou součástí nasazení aplikace je detailní znalost konfigurace prostředí, včetně nastavené operačního systému, obzvláště pro zajištění efektivní údržby celého prostředí a aplikace. Dokumentace instalačních procedur a jejich následná aplikace na prostředí nové poskytuje prostor k chybám a vyžaduje manuální zásahy, nehledě na časovou náročnost nové instalace a kontroly existujících prostředí. Pro usnadnění a automatizaci potřebných procedur existuje řada volně dostupných nástrojů, mezi nejpoužívanější se řadí:

- SaltStack

[92] Single point of failure - prvky, jejichž nedostupnost ovlivní dostupnost celého systému
[93] Virtual Router Redundancy Protocol
[94] Hot Standby Router Protocol, proprietární protokol od firmy Cisco

- Ansible
- Puppet
- Chef

Principem zmíněných nástrojů je definice výsledného prostředí, spíše než definice kroků, které se musí provést. Spíše tedy definujeme, že má specifický soubor existovat s konkrétním obsahem, než jak ho vytvořit a kontrolovat jeho atributy. Pro potřeby platformy OpenDVS byl vytvořen předpis pro aplikační konfiguraci v nástroji SaltStack, kdy konfigurace celého prostředí je definována sadou konfiguračních parametrů, označovanými jako *Pillars*.

```
nginx_install:
  pkg.installed:
    - sources:
      - nginx: salt://nginx/nginx-1.10.0-1.el7.ngx.x86_64.rpm

/etc/nginx/nginx.conf:
  file.managed:
    - template: jinja
    - user: root
    - source: salt://nginx/nginx.conf
    - context: {{ pillar['nginx'] }}

nginx:
  service.running:
    - enable: True
    - restart: True
    - watch:
      - file: /etc/nginx/nginx.conf
```

Výpis 9: Saltstack předpis pro konfiguraci aplikace nginx

Jelikož z hlediska bezpečnosti je vhodné fyzicky oddělit jednotlivé komponenty platformy, jakož i konfigurace čistého prostředí trvá jistou dobu a vyžaduje dostupnost všech běhových závislostí, nabízí se zde použití kontejnerizace. Zde je největším průkopníkem technologie *Docker*, která, za pomoci bezpečnostních mechanismů v jádře operačního systému Linux, umožňuje spouštění celého uživatelského prostředí operačního systému nad sdíleným jádrem (paravirtualizace). Výhodou technologie Docker je definování souboru *Dockerfile* s instalačními předpisy (zde již jako posloupnost prováděných kroků), kdy výstupem zde je binární obraz daného systému. Jelikož jádro systému Linux obsahuje i funkcionality pro emulaci rozdílných architektur, je

možné spouštět i kontejnery sestavené na jiné architektuře. Jelikož tento způsob nabízí efektivní možnost redistribuce celé platformy, byla taktéž vytvořena sada Docker předpisů pro snadné nasazení celé platformy.

6.2 Škálování SQL databází

Jestliže je nutné zajistit vysokou dostupnost databázových systémů, máme na výběr mezi *master-slave* konfigurací a *multi-master* konfigurací. První zmíněná konfigurace vyžaduje alespoň dva uzly, kdy pouze na jeden z nich (master) je možné zapisovat data. Na ostatních uzlech (slave) je možné číst data, za předpokladu, že nejsou součástí transakce[95]. Zde je také častá asynchronní replikace mezi master a slave uzlem. Druhá zmíněná konfigurace umožnuje zapisovat na všechny uzly současně za vynucení synchronní replikace a sníženého výkonu. Pro automatické zotavení pro havárii master uzlu (failover) lze konfigurací zajistit, který slave uzel má převzít jeho roli.

U pokročilých databázových systémů existuje řada metodik rozkládání zátěže, při využití inteligentního LB systému (například pgpool-II pro PostgreSQL) lze konfigurací zajistit právě předávání read-only požadavků na slave uzel celého systému. Jestliže je nicméně nutné škálovat databázový systém horizontálně, zmíněné přístupy zde nejsou velmi vhodné a mnohem efektivnější je zde *sharding*. Cílem daného přístupu je deterministicky rozprostřít data skrz několik uzlů, často i s jistou mírou replikace dat. Takováto konfigurace pouze předává požadavky na uzel, kde se data nachází, kdy determinismem je zde dosaženo hashováním specifické hodnoty prvku (např. jedinečný identifikátor záznamu). Přestože je tento přístup velmi častým pro nerelační databáze (NoSQL, například MongoDB), není vždy snadné stejný přístup aplikovat na databáze relační. Z open source databázových systémů lze zmínit platformu *Vitess*[96], která zmíněnou funkcionalitu aplikuje na MySQL databázi a je využívána ve velké míře u platformy YouTube. Jelikož platforma Vitess nevyžaduje modifikaci MySQL databáze, je i nasazení celého řešení relativně bezproblémové.

Přestože je zajištění vysoké dostupnosti velmi důležité pro efektivní chod celé platformy, je při nasazení vhodné vydefinovat výslednou architekturu nasazení dle cíleného rozsahu použití. I z hlediska výkonu clusterovaných databází je doporučeno nevyužívat synchronní replikace či inteligentního loadbalancingu[97] a pro nasazení je navrhována neclusterována verze MySQL databáze.

6.3 Testovací scénáře

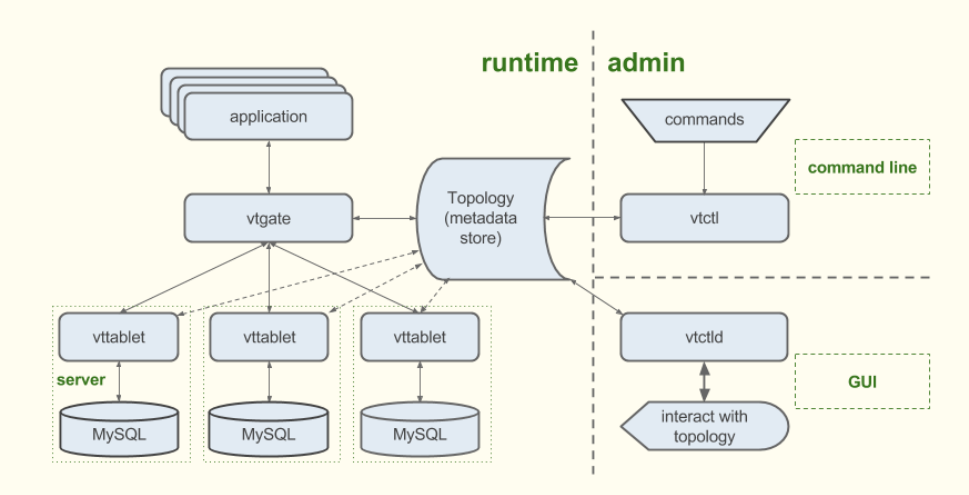
Pro úspěšné otestování funkcionality platformy byla vydefinována řada neformálních testovacích scénářů[98], které lze manuálně spustit. Každý testovací scénář je složen z testovaného souboru a

[95] Konkrétní implementace záleží na použitém databázovém systému

[96] https://github.com/youtube/vitess

[97] Přestože je dbán důraz na správné využívání transakčního kontextu, je v příapdě asynchronní replikace možnost neočekávaného chování platformy

[98] Spíše než formání testovací případy, pro které je definována specifická struktura, jsou tyto neformální scénáře vhodné pro vyzkoušení nabízené funkcionality



Obrázek 11: Architektura databázové platformy Vitess

definovaného chování. Lze tedy po nasazení platformy efektivně otestovat správnou funkčnost. Základní sada scénářů obsahuje otestování následující funkcionality:

- Detekce závislostí systému Maven (soubor *pom.xml*)
- Detekce závislostí systému npm (soubor *package.json*)
- Detekce zanořených závislostí v archivech typu ZIP
- Detekce zastaralé závislosti
- Detekce zranitelné závislosti
- Správné sestavení grafu komponent
- Úspěšná notifikace uživatele

Všechny testovací scénáře, včetně dokumentace platformy, jsou uloženy ve složce *docs* v hlavní složce zdrojového kódu. Potřebná dokumentace je vytvořená za pomoci značkovacího jazyka *Markdown*, který je hojně rozšířen mezi open-source komunitou. Při využití rozličných nástrojů (např. *GitBook*) lze vytvořenou dokumentaci taktéž vygenerovat do známých formátů (PDF, HTML, aj.), či využívat přímo z webového prostředí známých Git platforem (GitHub, GitLab, aj.). Díky využití verzovacího systému Git pro uložení dokumentace lze efektivně udržovat aktuálnost.

7 Plány do budoucna

Pro úspěšné využití platformy v enterprise sféře, jakožto i pro SaaS službu, je vhodné implementovat rozšiřující funkcionalitu. Přestože níže uvedená funkcionalita není součástí diplomové práce, je vhodné zmínit plány vývoje na následující měsíce.

7.1 Detekce balíčkovacích systémů

Velmi důležitou funkcionalitou pro efektivitu platformy je podpora širokého množství balíčkovacích systému. Pro prioritizaci programovacích jazyků bylo přihlédnuto k TIOBE indexu, tak i k statistice *GitHut 2*[99], agregující informace o používaných jazycích na platformě GitHub. Pro prioritizaci nových implementací balíčkovacích systémů bylo přihlédnuto k moderním trendům v oblasti využívaných platforem. Výsledný seznam obsahuje několik balíčkovacích systému, pro které bude vytvořena implementace, seřazených dle priority.

1. Gradle (Java)
2. PyPi (Python)
3. RubyGems (Ruby)
4. Docker
5. SBT (Scala)
6. Balíčkovací systémy operačních systémů (RPM/yum, deb/Aptitude, aj.)
7. NuGet (C#, .NET)
8. Composer (PHP)
9. CocoaPods (Switf, Objective-C)
10. Conan (C, C++)

Při implementaci nových sond je třeba brát důraz na sestavení grafové struktury, včetně transitivních závislostí. Taktéž by bylo vhodné vytvořit vlastní implementaci generátoru grafové struktury pro systém Maven, například jako plugin.

[99] Díky změně GitHub API je původní verze projektu GitHut na doméně *githut.info* zastaralá, byla tedy využita nová verze dostupná na adrese https://madnight.github.io/githut/

Pozice	Jazyk	Zastoupení	Roční změna
1	Java	15,568%	-5,28%
2	C	6,966%	-6,94%
3	C++	4,554%	-1,36%
4	C#	3,579%	-0,22%
5	Python	3,457%	+0,13%
6	PHP	3,376%	+0,38%

Tabulka 1: TIOBE index, Duben 2017
Popularita programovacích jazyků

Pozice	Jazyk	Trend
1	JavaScript	
2	Python	
3	Java	
4	PHP	+
5	Ruby	-
6	Go	+

Tabulka 2: GitHut 2 statistika, Q1 2017
Popularita programovacích jazyků

7.2 Definice externích repozitářů

Pro efektivní detekci neznámých závislosti je vhodné definovat proces zadání vlastních repozitářů. Z důvodu bezpečnosti není možné definovat tyto repozitáře globálně na každé sondovací instanci, ale spíše předávat speciální nastavení každému spuštění balíčkovacího nástroje. Také je na bezpečnost nutné myslet v kroku získávání komponent z existujících repozitářů, na která se mohou vztahovat rozdílná bezpečnostní pravidla. Je tedy pravděpodobně nutné vytvořit složený klíč *identifikátor-repozitář* u identifikovaných komponent. Zde je také nutné myslet na definici oprávnění repozitářů, která musí být vázána na specifický projekt spíše než na konkrétní uživatele.

7.3 Dynamické notifikace a integrace

Pro potřeby notifikace uživatelů a externích systémů bude implementován notifikační systém zmíněný v sekci 2.8. Pro možnost dynamické konfigurace notifikačních systémů bude definováno jednotné API, které bude možno stylizovat. Samotnou komunikaci pro notifikace nebude provádět jádro platformy, ale samostatná komponenta propojená pomocí direct queue. Pro striktní separaci databázových přístupů budou společně s notifikací do queue přidávány i specifické notifikační parametry. Toto řešení bude velmi efektivní z hlediska škálovatelnosti, opakování notifikace, či implementační změny (např. šablona emailu), nicméně již nebude tak efektivní v případě změny konfiguračních parametrů.

7.4 Externí sondy

Pro analýzu privátních aplikací, obzvláště v korporátní sféře, může být poskytování zdrojových kódu a binárních sestavení jistým bezpečnostním rizikem. Z tohoto důvodu by bylo vhodné vytvořit systém směrování a dedikování sond. Koncový uživatel by měl možnost spustit detekci komponent v kontrolovaném prostředí, kde by do platformy OpenDVS byly odesílány pouze metadata aplikace (detekované komponenty). Zatímco je velmi často tento princip definován pomocí vlastnoručně definovaného API, bylo by zajímavou volbou umožnit koncovým uživatelům

přímý přistup na MQ systém, kde by byl přístup zabezpečen pomoci oddělených MQ uživatelů a front.

7.5 Identity management

Pro potřeby identifikace uživatele je v platformě využíván protokol OpenID Connect. Přestože implementace za pomoci jediného zdroje identit je efektivní a snadná, pro autentizaci u veřejné služby pouze jeden IdP nemusí dostačovat. Pro veřejnou autentizaci existuje řada služeb, uživatel se může autentizovat pomocí účtů na službách jako Facebook, Google, GitHub či GitLab, a to převážně za pomoci zmíněného protokolu. Je důležité upozornit, že řada velkých IdP nemusí striktně dodržovat specifikaci daného protokolu[100] a je tedy třeba implementovat rozličné vyjímky.

Za pomoci použité komponenty *Spring Security OAuth* je možné za pomoci rozdílných autentizačních filtrů využívat několik IdP zároveň. První variantou tedy je definice rozhraní pro implementaci rozdílných poskytovatelů identit. Druhou variantou může být využití open-source IAM systémů (např. Keycloak od firmy RedHat), kde již za pomoci federace je umožněno využívání vícero IdP.

[100] Například implementace u Facebook či Google se mírně liší od zadané specifikace

8 Závěr

V rámci diplomové práce byla vyvinuta platforma pro snadnou detekci závislostí analyzovaných aplikací. Přestože byl brán velký důraz na korektní detekci definovaných balíčkovacích systémů, z technické stránky nebylo vždy možné striktně dodržet zadání. I přes překážky při vývoji byl vytvořen systém, který je možné efektivně využívat pro dané potřeby, to vše při zajištění zadané úrovně bezpečnosti.

Díky hybridní architektuře je možná velmi efektivní škálovatelnost jako v případě micro-services architektury, zároveň je také zabráněno zbytečné duplikaci dat mezi komponentami. Díky modulární implementaci je následná implementace dalších balíčkovacích sytémů a zdrojů informací velmi snadná. Z implementovaných funkcionalit platforma nabízí analýzu aplikací využívajících balíčkovací systém Maven a balíčkovací systém npm. Uživatel má také možnost zobrazení detailního přehledu analyzovaného projektu, jakožto i kontinuální analýzy pro případ nově vzniklého rizika. Pro autentizaci byl zvolen SSO protokol OpenID Connect, pro provoz platformy je tedy potřeba zajistit externí IdP poskytující integraci skrz zmíněný protokol.

Za dobu vývoje platformy bylo napsáno přes 6000 řádků kódu a více něž 100 tříd v jazyce Java, zároveň více než 2500 řádků kódu v jazyce JavaScript. Pro vývoj reaktivního frontendu bylo využito knihovny React a pro jádro platformy byl využit framework Spring Boot.

Platforma byla následně vydána pod licencí Apache ve verzi 2 a publikována na známý server GitHub.

Literatura

[1] DOWD, Mark., John MCDONALD a Justin. SCHUH. The art of software security assessment: identifying and preventing software vulnerabilities. Indianapolis, Ind.: Addison-Wesley, 2007. ISBN 978-0321444424.

[2] CVE-2016. National Vulnerability Database: Data Feeds [online]. 2017 [cit. 2017-04-17]. Dostupné z: https://nvd.nist.gov/vuln/data-feeds

[3] CVE-2017. National Vulnerability Database: Data Feeds [online]. 2017 [cit. 2017-04-17]. Dostupné z: https://nvd.nist.gov/vuln/data-feeds

[4] STEVENS, Marc, Elie BURSZTEIN, Ange ALBERTINI a Yarik MARKOV. The first collision for full SHA-1. [online], 2017. Dostupné také z: https://shattered.io/static/shattered.pdf

[5] Semantic Versioning 2.0.0. Semantic Versioning [online]. 2013 [cit. 2017-04-17]. Dostupné z: http://semver.org/spec/v2.0.0.html

[6] JAR File Specification. Java Documentation [online]. Oracle, 2014 [cit. 2017-04-17]. Dostupné z: http://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html

[7] Pattern: Microservice Architecture. Microservice Architecture [online]. 2015 [cit. 2017-04-17]. Dostupné z: http://microservices.io/patterns/microservices.html

[8] Inner-browsing extending the browser navigation paradigm. Mozilla Developer Network [online]. 2003 [cit. 2017-04-17]. Dostupné z: https://developer.mozilla.org/en-US/docs/Archive/Inner-browsing_extending_the_browser_navigation_paradigm

[9] The OAuth 2.0 Authorization Framework: RFC 6749. Internet Engineering Task Force (IETF) [online]. 2012 [cit. 2017-04-17]. Dostupné z: https://tools.ietf.org/html/rfc6749

[10] JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants: RFC 7523. Internet Engineering Task Force (IETF) [online]. 2015 [cit. 2017-04-17]. Dostupné z: https://tools.ietf.org/html/rfc7523

[11] JSON Web Signature (JWS): RFC 7515. Internet Engineering Task Force (IETF) [online]. 2015 [cit. 2017-04-17]. Dostupné z: https://tools.ietf.org/html/rfc7515

[12] JSON Web Encryption (JWE): RFC 7516. Internet Engineering Task Force (IETF) [online]. 2015 [cit. 2017-04-17]. Dostupné z: https://tools.ietf.org/html/rfc7516

[13] How Twitter Is Scaling. Waiming Mok's Blog [online]. 2009 [cit. 2017-04-17]. Dostupné z: https://waimingmok.wordpress.com/2009/06/27/how-twitter-is-scaling/

[14] STEFANOV, Stoyan. React: Up & Running: Building Web Applications. O'Reilly Media, 2016. ISBN 978-1491931820.

[15] YOUNG, Alex R. a Marc HARTER. Node.js in Practice. Manning Publications, 2014. ISBN 978-1617290930.

[16] SONATYPE. Maven: The Definitive Guide. O'Reilly Media, 2008. ISBN 978-0596517335.

[17] ODERSKY, Martin, Lex SPOON a Bill VENNERS. Programming in Scala. Third Edition. Artima, 2016. ISBN 978-0981531687.

[18] VENNERS, Bill a Vaughn VERNON. Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. Addison-Wesley Professional, 2015. ISBN 978-0133846836.

[19] RV, Rajesh. Spring Microservices. Packt Publishing, 2016. ISBN 978-1786466686.

[20] GUTIERREZ, Felipe. Pro Spring Boot: A no-nonsense guide containing case studies and best practices for Spring Boot. Apress, 2016. ISBN 978-1484214329.

[21] GUTIERREZ, Felipe a Tim MESSERSCHMIDT. Identity and Data Security for Web Development: Best Practices. O'Reilly Media, 2016. ISBN 978-1491937013.

A Přílohy na CD/DVD

Zdrojové kódy platformy včetně neformálních testovacích scénářů a anglické dokumentace jsou distribuovány na přiloženém CD.